ALGORITHMS FOR CONSTRAINT-BASED LEARNING OF BAYESIAN NETWORK STRUCTURES WITH LARGE NUMBERS OF VARIABLES

by

Martijn de Jongh

M.S. Computer Science,

Delft University of Technology, 2007

B.E. Electrical Engineering,

Technische Hogeschool Rijswijk, 2003

Submitted to the Graduate Faculty of

the School of Information Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2014

UNIVERSITY OF PITTSBURGH

SCHOOL OF INFORMATION SCIENCES

This dissertation was presented

by

Martijn de Jongh

It was defended on

April 22nd 2014

and approved by

Marek Druzdzel, PhD, School of Information Sciences

Stephen Hirtle, PhD, School of Information Sciences

Hassan Karimi, PhD, School of Information Sciences

Gregory Cooper, MD, PhD, Department of Biomedical Informatics

Denver Dash, PhD, Intel Labs

Dissertation Director: Marek Druzdzel, PhD, School of Information Sciences

ALGORITHMS FOR CONSTRAINT-BASED LEARNING OF BAYESIAN NETWORK STRUCTURES WITH LARGE NUMBERS OF VARIABLES

Martijn de Jongh, PhD

University of Pittsburgh, 2014

Bayesian networks (BNs) are highly practical and successful tools for modeling probabilistic knowledge. They can be constructed by an expert, learned from data, or by a combination of the two. A popular approach to learning the structure of a BN is the constraint-based search (CBS) approach, with the PC algorithm being a prominent example.

In recent years, we have been experiencing a data deluge. We have access to more data, big and small, than ever before. The exponential nature of BN algorithms, however, hinders large-scale analysis. Developments in parallel and distributed computing have made the computational power required for large-scale data processing widely available, yielding opportunities for developing parallel and distributed algorithms for BN learning and inference.

In this dissertation, (1) I propose two MapReduce versions of the PC algorithm, aimed at solving an increasingly common case: data is not necessarily massive in the number of records, but more and more so in the number of variables. (2) When the number of data records is small, the PC algorithm experiences problems in independence testing. Empirically, I explore a contradiction in the literature on how to resolve the case of having insufficient data when testing the independence of two variables: declare independence or dependence. (3) When BNs learned from data become complex in terms of graph density, they may require more parameters than we can feasibly store. I propose and evaluate five approaches to pruning a BN structure to guarantee that it will be tractable for storage and inference. I follow this up by proposing three approaches to improving the classification accuracy of a BN by modifying its structure.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

This dissertation is the culmination of my doctoral studies at the University of Pittsburgh. Finishing it was a task that I completed with the help and support of many, and I would like to take the opportunity to thank them here. I have spent the last six years working closely with Marek Druzdzel, my advisor, in the School of Information Sciences' Decision Systems Laboratory. I cannot thank Marek enough for his guidance, encouragement and wisdom. For teaching me not only about being a good scholar and researcher, but a thing or two about being a better person as well. I thank my committee, for their advice and feedback. I really enjoyed Dr. Hirtle's class on cognitive psychology, a field I was not yet familiar with. In a doctoral seminar taught by Dr. Karimi, I picked up a few skills and habits that proved very useful over the years. I am indebted to Dr. Cooper for his participation in my comprehensive exam. His assignment helped me realize what I can accomplish in a short time when I completely focus on a task. I got to known Denver better when we were working on a project together with Mark Voortman. Over the course of about ten months we spent many hours in weekly meetings discussing theory, practice, their differences, and paper deadlines. I thank dr. Chris White of DARPA for providing access to a Hadoop cluster for my work. During my stay in Pittsburgh I have made many friends, and am grateful for meeting all of them, and what they have meant for me. To name a few of them, I thank Mark for being an example to look up to. Tomek, for never a dull moment in a windowless office. Parot, for making our joint journey a memorable one. Kelly, for putting up with my whining, and Andrii, for our many interesting conversations and our unexpected adventures. I thank my parents, Martien and Jolanda, and my sister Nadine, for always being there for me, supporting me in whatever I do and wherever this may take me. Finally, I thank my wife Yan and daughter Scarlett for all their love and patience. I dedicate this work to them.

1.0 INTRODUCTION

Bayesian networks (BNs) [Pearl, 1988] are highly practical and successful tools for modeling uncertain domains. BNs and similar graphical models have enabled probabilistic models for many different types of tasks, in many different domains, and of many different magnitudes. The assumptions of BNs allow for efficient representation of probabilistic knowledge and efficient inference algorithms [Lauritzen and Spiegelhalter, 1988] for most cases, even though theoretically, both exact [Cooper, 1990] and approximate [Dagum and Luby, 1993] inference are NP-hard.

There are three main approaches to constructing Bayesian networks. The first approach focuses on constructing a model by consulting domain experts. This involves encoding and establishing relations among variables and estimating probabilities, which is usually time consuming. In the second approach, we construct a Bayesian network by learning the structure [Cooper and Herskovits, 1992, Spirtes et al., 1993] and the model parameters [Lauritzen, 1995] from a dataset. With the increased availability of data and computational power, the second approach, when available, is preferred. The third approach is a hybrid of the first two approaches. Here, two different routes can be taken to construct a model: 1) An expert constructs the graphical structure, and an algorithm is used to learn the parameters, and 2) The expert provides background knowledge to an algorithm, which then constructs a BN that satisfies the constraints set by the expert.

In recent years, we have been experiencing a data deluge. Datasets can be so large, they may not fit on one computer, and if they do, the exponential nature of BN algorithms might preclude practical and timely data analysis. One response to this problem has been to investigate and exploit parallelism. Clusters of computers are now commonly used to store large amounts of data and to process these data using parallel or distributed computing

paradigms. Popular approaches are based around the communication protocol standard MPI [Snir et al., 1995, Gropp et al., 1999], and the MapReduce framework [Dean and Ghemawat, 2004, White, 2012], popularized by publications of Google and the open source Hadoop framework, developed in part at Yahoo!

For both parallelization frameworks, work has been done on structure learning algorithms for Bayesian networks. Most of the work focused on creating parallel versions of the Bayesian search approach [Cooper and Herskovits, 1992]. There are examples of exact algorithms [Nikolova et al., 2009, Tamada et al., 2011], dividing up the search space [Xiang and Chu, 1999, Lam and Segre, 2002], local optimization [Liu et al., 2005], and particle swarm optimization [Sahin and Devasia, 2007]. All these examples make use of the MPI standard and the majority of the approaches are score-based, but a few are constraint-based [Spirtes et al., 1993, Tsamardinos et al., 2006, Nikolova and Aluru, 2011]. There are two MapReduce implementations of structure learning algorithms [Chen et al., 2011, Fang et al., 2013]. Both focus their parallelization efforts mostly on distributed computation of sufficient statistics necessary for the Bayesian score [Fang et al., 2013] or mutual information [Chen et al., 2011] calculation.

The MapReduce framework allows for computation on an enormous scale, providing the means for distributed computation and storage while shielding most technical details from the end-user, making it potentially a good platform for learning BN structures from large datasets. The two example algorithms from the literature allow for adequate support for analysis of data with many records, but both approaches require a large number of MapReduce jobs, dependent on the number of variables in the dataset. They have $O(n^4)$ [Chen et al., 2011] and $O(n^2)$ [Fang et al., 2013] worst-case requirements, and I found that the level of performance of the algorithms is no longer guaranteed when faced with datasets that are not necessarily "long" (records), but are "wide" (variables). I have investigated approaches based on the PC algorithm and have explicitly avoided dependence on the number of variables for the number of MapReduce jobs that the algorithms require, this to improve the ability of the algorithms to scale up to datasets with a larger number of variables.

1.1 CONTRIBUTIONS

In this dissertation, I propose two MapReduce versions of the PC algorithm, I address a contradiction in the literature on how to resolve the case of having insufficient data when testing the independence of two variables, I propose and evaluate five approaches to pruning a BN structure to guarantee that it will be tractable for storage and inference and I follow this up by proposing three approaches to improving the classification accuracy of a BN by modifying its structure.

While each of these contributions, discussed over the course of three chapters, addresses different problems, there is a common theme that connects them. In each case we will find that algorithm design decisions, be it for Bayesian network structure learning algorithms or for Bayesian network structure modification algorithms, have a significant impact on the end result. For instance, between the MapReduce algorithms I have evaluated there are significant differences in their ability to scale their structure learning performance to datasets with larger number of variables. Similarly, the default decision to be executed when there are not enough data samples to satisfy a recommended ratio of samples to contingency table cells, eventually impacts the tractability and quality of Bayesian inference run on a structure. A suboptimal choice here, may lead to networks that are too dense. This may cause potential overfitting problems when learning classifiers due to the increase in parameters. Alternatively, memory usage might increase beyond a single computer's ability to load the model in memory completely. Finally, when investigating potential solutions to this "network density" problem, I found statistically significant differences between different approaches aimed at controlling the number of parents of a network. The approaches I proposed to rank the parents of a node were presented the exact same networks, but some of them were unable to outperform a randomized approach on either of the tasks that I defined for the experiments, while other resulted in sparser networks without negatively impacting the performance. Although each of the contributions has in common that design decisions for the algorithms used for either learning or modifying Bayesian network structures can positively influence the tractability and quality of Bayesian network inference or Bayesian network structure learning, I have defined four separate hypotheses, each tailored to the

specific contribution. In the following three subsections, each representing a chapter in the dissertation (Chapters 5, 6, and 7 respectively), I elaborate on the work done and formulate the hypotheses that were tested.

1.1.1 MapReduce

I propose two MapReduce versions of the PC algorithm [Spirtes et al., 1993], aimed at solving an increasingly common case: data is not necessarily massive in the number of records, but more and more so in the number of variables. I evaluate the performance of these algorithms on a number of datasets, the largest having 4,933 variables, and compare their performance with two examples of MapReduce structure learning algorithms for Bayesian networks [Chen et al., 2011, Fang et al., 2013], and a single-processor version of the PC algorithm. I set out to test the following hypothesis:

- *H1: It is computationally feasible to learn Bayesian network structures from datasets that have very large numbers of variables using the MapReduce framework.*

1.1.2 Insufficient Data

When the number of data records is small, the PC algorithm experiences problems in independence testing. When there are insufficient data available for the independence tests, which are performed in the first phase of the algorithm, statistical errors may significantly influence the final BN structure. Here, data is considered insufficient when there are not enough data samples to satisfy a predetermined ratio of samples to contingency table cells.

In their book, Spirtes et al. [1993] suggest a ratio of 10 to 1 of samples to cells, ensuring more or less a minimum level of reliability for the independence tests. The advice is, that when the ratio is not satisfied, the test should not be performed and the variables connected by the edge should be considered dependent. Tsamardinos et al. [2006] state in their paper that they follow the advice by Spirtes et al. [2000, identical to 1993 version], but where Spirtes et al. recommend keeping an edge if there are to few records, Tsamardinos et al. recommend removing the edge. I am not sure whether this was a conscious decision or a misunderstanding, nevertheless it is interesting to explore this contradiction.

The classical χ^2 test of independence, used by the PC algorithm, defines as its null hypothesis that the variables are independent. Thus, variable dependence is something that requires support from the data. The recommendation of Spites et al. seems counterintuitive, since when we have sufficient data, before keeping an edge we have to disprove the null hypothesis. But, actually their advice is quite conservative. Networks that contain more arcs can represent more distributions, since they are less constrained by assumptions of independence. But, observing that many Bayesian networks in the real world, tend to be sparse, the conservative approach of Spirtes et al. might not necessarily be the best strategy. I have formulated the following hypothesis that I test in this dissertation:

- *H2: Faced with an insufficient data sample to total contingency table cell number ratio, when testing the independence of two variables connected by an edge, removing this edge results in better Bayesian network structures.*

1.1.3 Network Tractability

When Bayesian networks learned from data become complex in terms of graph density, they may require more parameters than we can feasibly store on disk or in memory, rendering them computationally intractable and, possibly from a practical point of view, useless. To remedy this problem, I propose and evaluate five approaches to pruning a BN structure to guarantee that it will be tractable for storage and inference, and I follow this up by proposing three approaches to improving the classification accuracy of a BN by modifying its structure. This work can be summarized in the following two hypotheses, which are tested in my dissertation:

- *H3: The tractability of inference of a Bayesian network can be improved by making structure modifications, without compromising its classification performance.*
- *H4: Well-chosen structure modifications can improve the classification accuracy of a Bayesian network.*

1.2 NOTATION

In the subsequent chapters I will use the following notation:

Variables and nodes are denoted by capital letters	:	X
Outcomes of variables are in lowercase	:	x
The cardinality of a variable X is denoted by	:	$Card\,(X)$
Sets of variables and nodes are in bold	:	$\mathbf{X}$
Power set of set $\mathbf{S}$	:	$\mathbf{PS}\,(\mathbf{S})$
The size of a set $\mathbf{S}$ is denoted by	:	$\lvert\mathbf{S}\rvert$
Parents of a Node X	:	$\mathbf{Pa}\,(X)$
Adjacent nodes of a node X	:	$\mathbf{Adj}\,(X)$
Test of independence between variables X and Y given set $\mathbf{Z}$	:	$I\,(X, Y\,\lvert\mathbf{Z})$
Graph	:	G
Vertex-set of Graph G	:	$\mathbf{V}\,(G)$
Edge-set of Graph G	:	$\mathbf{E}\,(G)$

Throughout this dissertation I use the terms *nodes* and *variables* to represent variables of a system modeled by a Bayesian network or variables of a dataset interchangeably.

1.3 ORGANIZATION OF THE DISSERTATION

The remainder of this dissertation is structured as follows. In Chapter 2, I cover the necessary background information. I describe previous work in Chapter 3 on the use of distance measures for quality assessment of structure learning algorithms, where I highlight the Hamming distance metric, which I use in the empirical evaluations of later chapters. In Chapter 4, I discuss related work on parallelizing structure learning algorithms. In Chapter 5, I propose two MapReduce versions of the PC algorithm, review two MapReduce algorithms from the literature, and evaluate these four algorithms. I conclude this chapter with a discussion on the feasibility of applying the MapReduce framework to BN structure learning algorithms.

In Chapter 6, I presents an inconsistency in the literature on how to act when faced with insufficient data when performing independence tests, and I evaluate empirically the performance of the proposed solutions of both views. In Chapter 7 I propose and evaluate five approaches to make a Bayesian network more tractable, and additionally, using the same methodology, I propose and evaluate three approaches that aim to improve the classification accuracy of a Bayesian network by making modifications to its network structure. I summarize my work in Chapter 8. Here, I discuss the outcomes of the experiments I have performed and if the data supports the hypotheses that I proposed in section 1.1 and I conclude with a discussion of possible future work.

2.0 BACKGROUND

In this chapter I cover the relevant background on Bayesian networks and distributed computing frameworks.

2.1 BAYESIAN NETWORKS

Bayesian networks [Pearl, 1988] (BNs) are efficient representations of joint probability distributions based on their factorization into products of conditional probability distributions. By an explicit representation of conditional independences among variables, they can significantly decrease the number of necessary parameters and make probabilistic modeling and inference in large problem domains feasible.

Any joint probability distribution can be factorized into a product of conditional probability distributions. This factorization can be represented by an acyclic directed graph (DAG). If a variable X_i is represented by a node X_i in the graph, the conditioning variables of X_i are represented as parents of the node X_i.

Typically, there are multiple DAG representations of a joint probability distribution. A trivial example of this is the case that represents a domain with no (conditional) independences whatsoever, and that can be described by the following factorization:

$$P(X_1X_2X_3\ldots,X_{n-1}X_n) = P\left(X_1\right)P\left(X_2\,|X_1\right)P\left(X_3\,|X_2X_1\right)\cdots P\left(X_n\,|X_{n-1,\ldots}X_3X_2X_1\right)\ . \tag{2.1}$$

This factorization can be represented by a completely connected directed graph. There are $n!$ possible orderings of n variables and, thus there are $n!$ possible DAGs that can represent this distribution.

If two variables X_i and X_j are (conditionally) independent of each other, neither of the two will appear in the other's conditioning set and, thus, neither can be a parent of the other in the DAG. There is thus no arc between the nodes X_j and X_i. If a DAG represents a factorization of a joint probability distribution, then this joint probability distribution can be recovered by applying the following equation

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\mathbf{Pa}(X_i)) \ , \tag{2.2}$$

where $\mathbf{Pa}(X_i)$ denotes the parents of X_i in the graph.

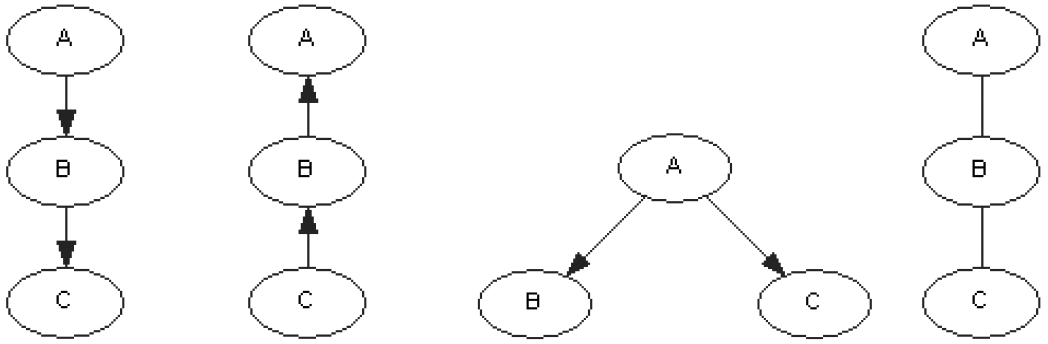
Different DAGs are able to accurately represent the same joint probability distribution. As an example, the $n!$ DAGs, with appropriate parameters calculated, are all equally capable of representing the original joint distribution, even though they have different structures. We can say that they are *structurally equivalent*, which is captured in the following [Chickering, 1995] definition:

> Two DAG structures are equivalent if the set of distributions that can be represented by one of the DAGs is identical to the set of distributions that can be represented by the other[1]

A DAG structure encodes a set of conditional independences. For another DAG to be considered equivalent, its structure must encode the same independences. Figure 2.1 shows three simple examples of DAGs that are all capable of representing the same distribution.

The existence of multiple structures that are equally capable of representing a joint probability distribution complicates the process of learning a structure from data. It is not possible, in general, to discover the orientation of all edges of a DAG, since the orientation of some edges may be immaterial to the set of independences that the graph represents. The inability to distinguish between graphs is known as *statistical indistinguishability*. [Spirtes

[1] I took this definition from Chickering [1995], although it was originally proposed by Verma and Pearl [1991] but differently phrased.



(a) (b) (c) (d)

Figure 2.1: Three DAGs that encode independence of A and C conditional on B (a through c). (d) shows a "pattern" representing these three graphs.

et al., 1993] Some substructures of graphs are recognizable. Figure 2.2 shows a DAG structure, known as a v-structure, which is uniquely identifiable and is a sole member of its equivalence class.

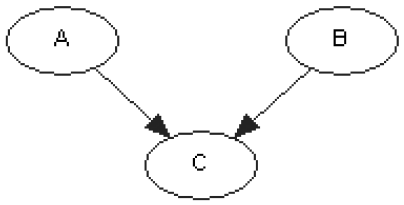


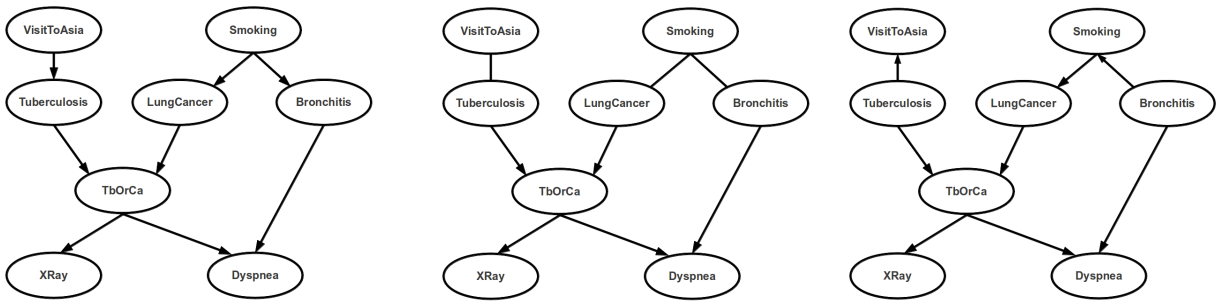
Figure 2.2: A v-structure

A v-structure is a triple of nodes, where two nodes, which are not directly connected, have edges directed into the third node. Finding v-structures is the key of the constraint-based search approach to structure learning [Spirtes et al., 1993]. v-Structures are also the key to defining equivalence classes over DAG structures.

> *Two DAGs are equivalent if and only if they have the same skeletons (both graphs have the same set of edges, disregarding exact edge orientation) and the same v-structures.* [Verma and Pearl, 1991]

An equivalence class $E(\mathcal{G})$ can be represented by a special graph structure called a partial directed acyclic graphs (PDAGs) or a *pattern*. [Verma and Pearl, 1991] The edges that belong to a v-structure are directed edges, and other edges, whose orientation typically cannot be determined, are represented by undirected edges. A pattern representing the DAGs in Figures 2.1(a) through 2.1(c) is shown in Figure 2.1(d).

Consider the Asia network [Lauritzen and Spiegelhalter, 1988]. Figure 2.3(a) shows the network, and Figure 2.3(b) shows the pattern that represents the equivalence class that the network belongs to. The top three edges are undirected in the pattern. The existence of undirected edges in the pattern means that the equivalence class has more than one member, exactly six in this case. Although there are eight possible combinations, two of them would cause a new v-structure to be created, which would bring the resulting network outside of its equivalence class. Figure 2.3(c) shows another member of the equivalence class Asia belongs to. Please note that some of the arcs are reversed compared to the original network in Figure 2.3(a).



(a) (b) (c)

Figure 2.3: The Asia network (a), its corresponding pattern (b) representing its equivalence class, and another instance (c) of Asia's equivalence class.

2.2 INFERENCE IN BAYESIAN NETWORKS

A Bayesian network efficiently represents a joint probability distribution. Its main use is to allow for inference to be performed on the variables that make up the joint distribution. Inference allows for interesting questions to be answer. In the medical domain, we might be interested to know which disease, given a set of observed symptoms, is the most likely cause of the symptoms. Conversely we can infer which symptom is the most informative when diagnosing a disease. Additionally, we can define a cost model on the tests one can perform on a patient. This allows us to find the most cost effective way to diagnose a disease or distinguish between multiple diseases.

In the 1970s, pioneering work was done on probabilistic inference, but this work was hindered by the computational complexity of working with full joint probability distributions. Table size and calculation time are both exponential in the number of variables, which prohibited the use of probabilistic models beyond simple toy problems. It was not until the 1980s with the appearance of Bayesian networks that larger models became feasible. Although the space for the model and time required for inference are still exponential in the worst case, algorithms have been developed that make inference feasible on large networks.

We can divide the approaches to Bayesian network inference into two groups: 1) exact inference and 2) approximate inference. The exact algorithms calculate probabilistic queries without error. The second group of algorithms usually trade accuracy for algorithm simplicity. There are cases where inexact algorithms give better time performance than exact methods without sacrificing too much accuracy, on others, exact methods will outperform the inexact counterparts.

2.2.1 Exact Inference

In general, the type of queries we are interested are in the form of $P(\mathbf{A}|\mathbf{B})$, were both $\mathbf{A}$ and $\mathbf{B}$ can be sets of variables. We call the conditioning set $\mathbf{B}$ evidence, variables for which we have observed their current states. The set $\mathbf{A}$ consists of the query variables, i.e., the variables that we are interested in. If we just have a joint probability distribution consisting

of $P(\mathbf{A}, \mathbf{B}, \mathbf{C})$, where $\mathbf{A}$ are the query variables, $\mathbf{B}$ are the evidence variables, and $\mathbf{C}$ are neither evidence nor query variables, we can perform inference in the following way:

$$P(\mathbf{A} \,|\mathbf{B}) = \frac{\sum_{\mathbf{C}} P(\mathbf{A}, \mathbf{B}, \mathbf{C})}{\sum_{\mathbf{A},\mathbf{C}} P(\mathbf{A}, \mathbf{B}, \mathbf{C})} \,. \tag{2.3}$$

This approach does not scale, as we need exponential space for the probability distribution and need exponential time for the calculation. The graph structure of a Bayesian network aids algorithms that cleverly reuse computation results to allow for fast and efficient inference most of the time.

2.2.1.1 Variable Elimination One approach that is an improvement over the naive brute force calculation of probabilities is called variable elimination. Using the chain rule and the Bayesian network we can break up a joint probability distribution into a product of conditional probability distributions. Breaking up de the joint distribution into the product of conditionals does not solve our complexity problem just yet, but it allows us to push in summations which will save us a considerable amount of computation. Consider again the Asia network from Figure 2.3(a). The joint distribution of the network is factored in the following way.

$$\begin{aligned} &P(A, S, T, L, B, O, X, D) = \\ &P(A)\, P(S)\, P(T \,|A)\, P(L \,|S)\, P(B \,|S)\, P(O \,|T, L)\, P(X \,|O)\, P(D \,|B, O) \,. \end{aligned}$$

If we want to calculate $P(L \,|X = \mathit{true})$ (Probability of lung cancer given a positive X-ray result) we will get the following equation:

$$\begin{aligned} &P(L \,|x) = \\ &\alpha \sum_a P(a) \sum_t P(t \,|a) \sum_s P(S)\, P(L \,|s) \sum_b P(b \,|s) \sum_o P(o \,|t, L)\, P(x \,|o) \sum_d P(d \,|b, o) \,. \end{aligned}$$

Where α is the normalization factor $P(x)$. The last term $\sum_d P(d \,|b, o)$ can be eliminated immediately, since it simply sums to 1, which leaves us with

$$P(L \,|x) = \alpha \sum_a \underbrace{P(a)}_{A} \sum_t \underbrace{P(t \,|a)}_{T} \sum_s \underbrace{P(S)}_{S} \underbrace{P(L \,|s)}_{L} \sum_b \underbrace{P(b \,|s)}_{B} \sum_o \underbrace{P(o \,|t, L)}_{O} \underbrace{P(x \,|o)}_{X} \,.$$

The conditional probabilities are usually represented as *potentials*. Potentials are non-negative functions, they thus more general than probability distribution functions, where besides the requirement of nonnegative entries, the probabilities are required to sum to one. This simplifies the operations on the factors to do the necessary calculations. Effectively, the factors are simply tables with entries for all possible combinations of the variables they contain. For variable elimination, three operations are defined for the factors: 1) Summation, 2) Product, 3) Observation of Evidence. When we apply summation, we sum out one or more variables from the factor. One can visualize this for a two-dimensional factor by summing all rows of a matrix, leaving just one row that contains the sums of all (original) columns. A product is defined over two factors. The result is a new factor which variable set is a union of the variable sets of the factors the product was calculated over. To calculate the entries of the new factor the entries of the two input factors are matched on their variable values:

$$F_{OTLX}\left(o_1, t_2, l_3, x_2\right) = F_{OX}\left(o_1, x_2\right) * F_{OTL}\left(o_1, t_2, l_3\right) \ . \tag{2.4}$$

Here the O values needs to be equal for factors F_{OTL} and F_{OX} to calculate the entries for the new factor F_{OTLX}. A scalar product, with α is calculated by multiplying α on all elements of the factor. When we observe evidence for a variable (Here, X), we set to '0' all entries of (at least) one factor containing the variables that are not consistent with the evidence. Minimally, this can be implemented as a product with a factor with one variable which has a '1' for the evidence state and a '0' for the others, which makes the third operation unnecessary. With these operations we can now calculate the inference $P\left(L \,|x\right)$ by applying multiplication and summation repeatedly. The normalization constant α can be acquired (inverted) by summing out the variable L from factor $F_{Lx}\left(L, x\right)$, the end result of summing and multiplying the factors.

The efficiency of the algorithm depends on the order of variables being summed out. In a bad ordering, we end up with larger factors that contain more variables. Since the factors must contain an element for all possible combinations of its variables, this makes the variable elimination algorithm exponential in the largest factor. Finding an optimal ordering is a NP-complete problem, hence usually heuristics are used.

2.2.1.2 Junction Tree Algorithm

By using factors, variable elimination is preventing unnecessary computations, making probabilistic inference usually more efficient than the brute force approach. A key problem with variable elimination and the brute force approach is that for every inference you effectively have to start over from scratch. A common Bayesian network inference task is to calculate the posterior distribution for all non-evidence nodes. This would require to perform variable elimination n times if there are n non-evidence variables. The junction tree algorithm [Lauritzen and Spiegelhalter, 1988] builds a special data structure which allows for easy calculation of all posteriors for all non-evidence variables.

The junction tree data structure is constructed over a few steps, please refer to [Huang and Darwiche, 1996] for a very clear and in-depth explanation of each step.

1. First a copy of the BN is made and all directed edges in this copy are made undirected. Using the original graph, the undirected copy is moralized. For each child node all parents are "married" by adding undirected edges between all pairs of parents of the child.
2. The moral graph is then triangulated. This is accomplished by adding an edges between pairs of nodes that are part of a loop that contains four or more nodes. The process continues until the are no more loops of length four or more.
3. The triangulated graph is then used to create clique nodes. These cliques contain multiple nodes. The cliques are created by removing nodes from the triangulated graph and noting the nodes which were connected to the removed node. Cliques that are subsets of other cliques are removed.
4. The created clique nodes are formed into the final junction tree structure by ensuring that neighboring cliques always have a subset of nodes in common and that for any subset of variables there exists only one path in the junction tree.

There usually is more than one way to translate a Bayesian network into a junction tree. One factor is the order of removing the variables from the triangulated graph, known as the elimination order. Different elimination orders create different sets of cliques which can can have different numbers of variables. The potential table that comes with a clique increases in size exponentially with the number of variables.

Finding the optimal elimination order, however, is a NP-complete problem [Cooper, 1990]. Usually heuristics are used to find elimination orders that are adequate.

After constructing the junction tree, it is initialized by multiplying the conditional distributions of the Bayesian network on the cliques, which when created are initialized with identity potential tables (all ones). Some of the distributions may fit into multiple cliques, if the variables in the conditional distribution match variables in multiple cliques, but the distributions should only by multiplied once, but it is immaterial on which clique this is done. Evidence is set in the same way as variable elimination. It is sufficient to set evidence for a variable on only one clique.

Inference in junction trees works by passing messages among the nodes. These messages propagate the evidence that was introduced in the clique potentials of the tree structure. After the message passing scheme has been completed all the cliques are updated with the evidence and, the reason why junction trees are so convenient for BN inference, each clique potential contains the joint distribution of the variables that are contained within the clique. At this stage, computing the posterior distribution of any variable reduces to finding a clique that contains this variable and marginalizing out all other variables in the clique. Clique potentials are orders of magnitude smaller than the full joint distribution, thus calculations are much more efficient. The message passing scheme works as follows:

1. Pick any clique in the junction tree
2. Recursively collect the messages from all other cliques and incorporate the messages in this clique
3. Calculate a message from the clique and distribute it recursively to all other cliques

After this two-phase scheme finishes, all cliques will contain the joint distributions of their variables.

2.2.2 Approximate Inference

Even with its enormous space-time improvements, there are still cases where using exact inference methods such as the junction tree algorithm on certain models, would still require an undesirable amount of space. Approximate algorithms have been developed as a trade

off between having the exact answer and the amount of space or time necessary to compute the answer. Next, I discuss two approaches to approximate inference.

2.2.2.1 Loopy Belief Propagation LBP [Murphy et al., 1999], is based on a precursor [Kim and Pearl, 1983] to the Junction tree algorithm [Lauritzen and Spiegelhalter, 1988]. Instead of having a secondary structure to perform the message passing procedure on, the plain Bayesian network structure is used. Between nodes, messages are calculated and are passed around. This algorithm is exact when the BN is a tree. If the BN does not satisfy this requirement the algorithm will not be able to calculate the exact solution. LBP has a slightly modified message passing scheme, compared to the junction tree algorithm. Normally a node needs to first collect all messages from its neighbors before passing its own message. When there are (undirected) loops in the graph, for instance, due to nodes with multiple parents that share a common ancestor, some nodes may never be able to pass messages since they will be waiting for other messages from nodes in the loop, which in turn are also waiting for messages from nodes in the loop, causing a deadlock.

LBPs solution is for nodes to collect all available messages and then to send their own message. Since the messages now can be incomplete, the process is repeated for all the nodes over many iterations until the distributions in the network converge to a stable state. The algorithm does not guarantee convergence, but tends to work quite well in general [Murphy et al., 1999].

2.2.2.2 Stochastic Sampling is a different approach to approximate inference. We use the Bayesian network to generate data samples. These samples are then used to estimate the query probability using the usual maximum likelihood estimation approach. Assume we generate some samples for the Asia network. We could end up with the following samples:

The simplest way to generate samples like these is with a method known as Probabilistic Logic Sampling [Henrion, 1988]. We sort the nodes topologically (parents appear before all of their children). We are then guaranteed to start with a node that is parentless. Such a node has a prior distribution instead of a conditional distribution (if a node has parents). Using a random number generator we can sample a value for this node. This value is then set

Table 2.1: Data Sample from the Asia Network

VisitToAsia	Tuberculosis	Smoking	LungCancer	TbOrCa	XRay	Bronchitis	Dyspnea
NoVisit	Absent	NonSmoker	Absent	Nothing	Normal	Present	Present
Visit	Absent	NonSmoker	Absent	Nothing	Normal	Absent	Present
NoVisit	Absent	NonSmoker	Absent	Nothing	Normal	Present	Present
NoVisit	Absent	Smoker	Present	CancerORTuberculosis	Abnormal	Absent	Present
NoVisit	Absent	Smoker	Absent	Nothing	Normal	Present	Present
NoVisit	Absent	Smoker	Present	CancerORTuberculosis	Abnormal	Present	Present
NoVisit	Absent	NonSmoker	Absent	Nothing	Normal	Absent	Absent
NoVisit	Absent	NonSmoker	Absent	Nothing	Normal	Absent	Absent
NoVisit	Absent	NonSmoker	Absent	Nothing	Normal	Present	Present
NoVisit	Absent	Smoker	Present	CancerORTuberculosis	Abnormal	Present	Present

for this node and we move on to the next node. Nodes that have parent are conditioned on the values of their parents and then sampled in the same way as the parentless nodes. The process continues until we have sampled values for all nodes in the network. This process is continued until we have the number of network samples we need.

If we want to estimate $P\left(Dyspnea = Present\right)$ we count the number of times the node $Dyspnea$ has the value *present* in the sample and divide this number by the total number of samples. Our estimate would then be $\hat{P}\left(Dyspnea = Present\right) = 0.8$, which still is quite far from the actual probability of 0.436. With more samples available to us our estimate would improve.

Up to this point the procedure is very simple and is quite space efficient, with the only requirement being that sufficient samples were generated to allow for accurate estimates. When we introduce evidence to our queries, we have to change the procedure to still be able to acquire accurate estimates. If we set evidence, we must require that all samples of the Bayesian network that we generate are consistent with this evidence. The simplest approach to comply with this requirement is called Rejection Sampling. Here the procedure is as before, but all samples that are inconsistent with the evidence are discarded. This is a simple and a valid solution, but its efficiency now depends on the probability of the evidence. Since all samples must be consistent with the evidence, the proportion of valid samples vs the total number of generated samples is equal to the probability of evidence. For example, if we have a probability of evidence $P\left(e\right) = 0.0001$, for every valid sample we will roughly discard 10000 samples. With even more improbable evidence it might take a long time before

we have gathered a sufficient number of samples to properly estimate the probability of the queries we are interested in. More sophisticated sampling schemes exist that improve on the basic procedure.

We can improve efficiency by not discarding samples but by weighting them appropriately. The approach, known as likelihood weighting [Fung and Chang, 1990, Shachter and Peot, 1990, Shwe and Cooper, 1991], weights each sample by the likelihood of the evidence. The process is similar to probabilistic logic sampling, the main difference is that we no longer count the occurrences of node states but sum the weights for corresponding samples, i.e., instead adding a '1' to the count, we add a fraction w.

$$w\left(\mathbf{E}=\mathbf{e}\right)=\prod_{i=1}^{n}P\left(E_i=e_i\left|\mathbf{Pa}\left(E_i\right)\right.\right)\ . \tag{2.5}$$

Likelihood weighting improves sampling performance, but is still susceptible to problems with unlikely evidence.

The previous sampling methods generate each sample from the same starting point. Other methods exist that describe the sample space as a state space that the algorithm traverses. These are known as Markov Chain Monte Carlo (MCMC) algorithms. Two example MCMC algorithms are 1) Gibbs sampling [Pearl, 1987], and 2) Metropolis-Hastings sampling [Metropolis et al., 1953]. Both methods are used when it is difficult to sample directly from the probability distribution we are interested in. Gibbs sampling is a special case of Metropolis-Hastings, and is frequently used for inference in Bayesian networks. Both sample from a helper distribution that needs to be proportional to the distribution of interest. The general procedure of MCMC algorithms is:

The main idea of these algorithms is that we continue generating samples from the Markov chain (which is formed by randomly traversing through the sample space), until the sampler has reach its stationary distribution. If the helper distribution was properly chosen, this stationary distribution will be the distribution that we are interested in.

To generate samples for a Bayesian network using Gibbs sampling, which is much simpler than the full Metropolis-Hastings algorithm, see Algorithm 2. Eventually the Gibbs sampler will reach its stationary distribution, and at that point we can extract samples to use for inference calculations.

Algorithm 1 MCMC

1: Pick a starting point **x**
2: **while** the required number of samples has not yet been generated **do**
3: Use the helper distribution to generate the next point **x'**
4: Calculate the likelihood L of **x** and **x'**
5: **if** $L(\mathbf{x'}) > L(\mathbf{x})$ **then**
6: accept **x'**
7: **else**
8: accept **x'** with probability p proportional to the likelihood ratio
9: **end if**
10: **end while**

Algorithm 2 Gibbs Sampling

1: Find a starting sample for the Bayesian network (any of the earlier discussed sampling methods would be a possible candidate).
2: **while** the required number of samples has not yet been generated **do**
3: Pick one of the non-evidence variables.
4: Using its conditional distribution and the variable instantiations of its parents (if it has any), randomly generate a new state for this variable.
5: **end while**

MCMC algorithms are convenient solutions for solving difficult problems such as high dimensional integrals and probabilistic inference, as long as appropriate helper distributions are available (especially for Metropolis-Hastings). The algorithms do have a few drawbacks associated with them: 1) The samples are not i.i.d. (independent identically distributed) and to regain a level of independence among the samples the majority of the samples has to be discarded. A common approach is to keep 1 out n samples, where n might depend on the particularities of the problem. 2) Our starting position might be far away from the stationary distribution where the algorithm will end up eventually. This issue is usually handled by first discarding all samples after starting the algorithm until the sample distribution has moved closer to the stationary distribution (this is known as the *burn-in* period). Typically a fixed number of samples is discarded, after which samples will be kept for the query estimation.

Another class of approximate inference algorithms is based on the principle of importance sampling. In general importance sampling is used for estimating high dimensional integrals.

$$V = \int_{\Omega} f(\mathbf{X})\, d\mathbf{X} \,, \tag{2.6}$$

Where $f(\mathbf{X})$ is a function over domain Ω in $\Re^n$. The idea of importance sampling is to find an importance function $I(\mathbf{X})$, a probability distribution that is easy to sample from. Applying the function to Equation 2.6 gives us:

$$V = \int_{\Omega} \frac{f(\mathbf{X})}{I(\mathbf{X})} I(\mathbf{X}) \,. \tag{2.7}$$

Although algebraically identical to the original equation this now can be considered as an expected value calculation. Assuming that V exists and that the importance function $I(\mathbf{X}) > 0$ for any $\mathbf{X}$ in Ω, we can start generating samples from $I(\mathbf{X})$ and construct an estimator $\hat{V}$:

$$\hat{V} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{I(X_i)} \,. \tag{2.8}$$

Under certain assumptions, when N becomes large, $\hat{V}$ will converge to V. These assumptions are [Geweke, 1989, Yuan and Druzdzel, 2006]:

- $f(\mathbf{X})$ is proportional to a proper probability density function defined on Ω

- $\{X_i\}_{i=1}^{\infty}$ is a sequence of i.i.d. random samples, the common distribution having a probability density function $I(\mathbf{X})$
- The support of $I(\mathbf{X})$ includes Ω
- V exist and is finite

The main challenge of importance sampling is finding a good importance function, although the optimal importance function has been proved [Rubinstein, 1981] to be:

$$I(\mathbf{X}) = \frac{f(\mathbf{X})}{V} . \tag{2.9}$$

This ends up being the posterior distribution of what we are interested in, making it not a feasible choice. It is however, useful as a guide to finding a good importance function. The closer a candidate function approximates the optimal function, the better our importance sampling procedure will perform.

There is a sizable body of work on applying the principle of importance sampling to Bayesian network inference [Shachter and Peot, 1989, Fung and Del Favero, 1994, Hernandez et al., 1998, Ortiz and Kaelbling, 2000, Cheng and Druzdzel, 2000, Yuan and Druzdzel, 2006], Yuan and Druzdzel [2006] divide the approaches into 3 groups. The first group proposes to use a prior distribution of a Bayesian network, which is not very effective once improbable evidence needs to be considered when performing inference. Both probabilistic logic sampling and likelihood weighting can be considered to be part of this group.

The second group proposes to learn and improve the importance function as the sampling process continues. Starting out with an initial function, it is periodically revised based on the samples generated thus far. One example of such an approach is the AIS-BN algorithm by Cheng and Druzdzel [2000]. The importance function is initialized with a prior distribution. The algorithm regularly updates the importance function using two heuristics: 1) it initializes the probability distributions of the parents of evidence nodes to uniform distributions, 2) very small probabilities present in conditional probability distributions that are part of the importance function are slightly increased to a preset value ϵ. Samples are then drawn and used to estimate the new importance function. Iteratively the importance function will improve and approach the optimal function, thereby better samples will be generated to approach the correct answer of the query to be answered.

The third group calculates an importance function using both a prior distribution and the evidence set provided for the inference. One example algorithm is the EPIS-BN algorithm by Yuan and Druzdzel [2006]. The algorithm applies loopy belief propagation to calculate an approximation of the optimal importance function. Next, as in the AIS-BN algorithm, all probabilities in the importance function with a value smaller than ϵ are replaced by ϵ. At this point the importance function is ready to be used for generating samples for the estimation of the inference query.

2.3 LEARNING BAYESIAN NETWORKS FROM DATA

Although Bayesian networks can be constructed by eliciting information from domain experts, the availability of data sets created the need for directly harnessing these data to create Bayesian networks in an automated fashion. Statistical procedures and algorithms exist that can learn both the structure and the parameters.

2.3.1 Parameter Estimation

If we assume we already have the Bayesian network structure, we can learn the model parameters if we have a data set available. The appropriate algorithm to use depends on the data set. If the data set has missing values we must treat the data in a different manner than if it would be complete.

2.3.1.1 Complete Data When we have complete data available, we can apply basic maximum likelihood estimation to determine the parameters for each of the variables. Maximum likelihood estimation is a common approach to estimate model parameters. One starts with a model with parameters $\boldsymbol{\theta}$ and a dataset $\mathbf{D}$ and defines a likelihood function:

$$L\left(\boldsymbol{\theta}\,|\mathbf{D}\right) = \prod_{i=1}^{n} P\left(d_i\,|\boldsymbol{\theta}\right) \ . \tag{2.10}$$

The data and the parametric model are held constant while we vary the model parameters. This will influence the value of the likelihood function and our final goal is to maximize the

likelihood function output by choosing the optimal values for $\boldsymbol{\theta}$. For computational reasons we usually use the log of the likelihood function for the optimization process:

$$log\left(L\left(\boldsymbol{\theta}\,|\mathbf{D}\right)\right) = \sum_{i=1}^{n} log\left(P\left(d_i\,|\boldsymbol{\theta}\right)\right) \ . \tag{2.11}$$

Taking the log of the likelihood function allows us to turn the product into a sum, which is useful when computing the result of the likelihood function on a computer where the precision for computation is limited to a certain number of digits.

For most parametric distribution we can derive the optimal maximum likelihood estimator (MLE) setting the derivative of the (log)likelihood function to 0 and to solve for $\boldsymbol{\theta}$. Bayesian networks are usually modeled using (conditional) multinomial distributions. For the multinomial distribution the MLE for each of the variable states is equal to the proportion of the state counts in the dataset, which is very convenient:

$$\hat{P}\left(X_1 = x\right) = \frac{N\left(X_1 = x\right)}{N} \ , \tag{2.12}$$

where N is the total number of samples in $\mathbf{D}$. For conditional multinomials we apply Bayes rule:

$$\hat{P}\left(X_1 = x_1\,|X_2 = x_2\,\right) = \frac{N\left(X_1 = x_1, X_2 = x_2\right)}{N\left(X_2 = x_2\right)} \ . \tag{2.13}$$

Using the counts of a complete dataset we can apply maximum likelihood estimation to estimate all parameters of a Bayesian network when we have the BN structure already available to us. To acquire all parameters we need to estimate prior distributions $P\left(X\right)$ for variables X without parents and conditional distributions $P\left(X\,|\mathbf{Pa}\left(X\right)\right)$ for variables X with parents $\mathbf{Pa}\left(X\right)$. When we do not have a complete dataset we must apply different approaches to properly deal with the missing values in the dataset.

2.3.1.2 Incomplete Data Having missing values in the dataset makes maximum likelihood estimation no longer a viable option. If some samples have missing values, i.e., some variables do not have a value, we could remove these samples from the dataset to make it complete again. This, however, might influence the probability estimates of the other variables, and if we want to study the effect of a latent variable, i.e., having no data available for a variable at all, we must use a parameter learning approach that has the ability to model a variable for which we do not have any data.

Data might be missing for different reasons, there is the latent variable case, where we do not have any data, data may be missing randomly, or there may be systematic reasons for a dataset to have missing values. We can classify missing data in the following way:

1. Missing Completely at Random (MCAR)

   In this case values are missing due to circumstances unrelated to any of the variables in the dataset, which we can formalize by stating that we have completely observed variables $\mathbf{X}$, and variables with missing values $\mathbf{Y}$, the probability of a value i of variable Y_j being missing (y_{ij}) can be modeled as $P(y_{ij}|\mathbf{X},\mathbf{Y}) = P(y_{ij})$. This indicates that the missingness of value y_{ij} is independent of the variable sets $\mathbf{X}$ and $\mathbf{Y}$. An example of a MCAR case are latent variables, since we do not have any data in the first place, the missingness of the data does not depend on specific values of the observed data that we do have access to. Another example might be certain types of sensor failures, accidentally destroying a lab sample, or people dropping out of research study due to reasons unrelated to what is being studied such as leaving the trial due to moving across the country or dying in a plane crash.
2. Missing at Random (MAR)

   When a dataset has variables that are MAR, the missingness of the values of these variables depends only on the other, completely observed variables in the dataset. Formally we can describe the relation between the missingness of values and the completely observed variables as $P(y_{ij}|\mathbf{X},\mathbf{Y}) = P(y_{ij}|\mathbf{X})$. The missing values are distributed conditioned on the values of the observed variables. Typical examples of MAR situations are 1) Distinct subgroups in a population declining to answer certain survey questions more often than other groups, 2) Depending on the results of some medical tests, such as

the HIV test, when the test turns out to be positive, a secondary test is usually administered that is not based on the same chemicals/principle as the first test. This is done to confirm the outcome of the first test and to decrease the likelihood of the possibility that the first test result was a false positive. Assuming most tests come out negative, only one test result will be available for most people in a dataset, and having the outcome of the second test result missing only depends on the outcome of the first test.

3. Missing Not at Random (MNAR)

   If the missingness of values depends on the variable with the missing value itself, regardless if it depends on the completely observed variables, the values are missing not at random. Formally, the probability of missingness is then defined by $P(y_{ij} | \mathbf{X}, \mathbf{Y})$. An example of MNAR is asking people for their salary on a survey and finding that people that are richer (or perhaps poorer) than most are not filling out this information.

Having variables in a dataset that have missing values that are MNAR is problematic. For proper modeling of the data, the mechanism responsible for the missing values needs to be modeled explicitly. The MAR and MCAR conditions are easier to cope with. A simple solution for MCAR missingness is the remove the samples with missing values, and then perform the usual estimation procedures. This is not possible for MAR missingness, here statistics can be estimated first for each conditioning group and then a weighted average can be computed. For Bayesian networks a common algorithm used when datasets have missing values is the *Expectation Maximization algorithm*, it assumes missing values are MAR and is a very popular algorithm for this purpose. I discuss this algorithm in the following subsection.

2.3.1.3 EM Algorithm The EM algorithm [Dempster et al., 1977, Lauritzen, 1995] is a general purpose algorithm for parameter estimation when the available dataset has missing values. The algorithm consists of two steps: 1) the expectation step and 2) the maximization step. Essentially, in the first step the algorithm fills in the gaps in the dataset using the current guess for the model parameters. In the second step, having filled in the gaps in the dataset a maximum likelihood estimation is performed to estimate new model parameters. The algorithm now again switches to step 1 and uses the new parameters to reestimate the missing values in the dataset after which the algorithm executes step 2 again. The algorithm

continues to iterate between step 1 and step 2 until the model likelihood converges, i.e., no longer increases. The EM algorithm is used for many different types of probabilistic models and exists in many forms. When used for Bayesian networks the algorithm works very nicely in conjunction with the junction tree algorithm to acquire the necessary counts used in the parameter estimation process.

During the expectation step of the algorithm we calculate expected counts for the dataset. When we have a complete dataset we can simply add 1 to the total count for each configuration of values, but in a incomplete dataset, when we come across a missing value we do not add a 1. We add a proportion that is equal to the probability of the missing value having the value of interest, conditioned on values of the other variables in the current sample. For Bayesian networks we need either prior distributions or conditional distributions. Calling all other variables in the sample the evidence e and the variable of interest, then the probability of X having value j is $P(x_j \,|e\,)$. Calculating the counts for a parentless node then becomes:

$$N(X=j) = \sum_{i=1}^{n} \left(I(X_i = j) + I(X_i = ?)\, P(X_i = j \,|e\,) \right) \ , \tag{2.14}$$

where I is an indicator function that is value 1 when its condition is met and 0 otherwise. Calculating the new probabilities for the prior distribution is accomplished by dividing the counts for the individual states j of the variable X by the total number of samples N in the dataset, i.e.,

$$P(X=j) = \frac{N(X=j)}{N} \ . \tag{2.15}$$

For conditional distributions $P(X \,|\mathbf{Pa}\,(X)\,)$ we require two types of counts: 1) counts for the joint distribution of X and its parents, $N(X, \mathbf{Pa}\,(X))$, and 2) counts for the joint distribution of X's parents $N(\mathbf{Pa}\,(X))$. To estimate the conditional probabilities we make use of Bayes' rule:

$$P(X \,|\mathbf{Pa}\,(X)\,) = \frac{N(X, \mathbf{Pa}\,(X))}{N(\mathbf{Pa}\,(X))} \ . \tag{2.16}$$

It suffices to calculated only all possibilities of $N(X, \mathbf{Pa}\,(X))$, by summing them all together we will get $N(\mathbf{Pa}\,(X))$. We will need counts for every combination of values of the node and its parents. When encountering missing values in the dataset, as with the prior distribution case we replace the 'hard' count of 1 for the combination by the probability of the variables

having the value combination given the observed values of the other variables in the sample. Calculating the probabilities of the required joint distribution is greatly simplified by the property of junction trees that a node must always be cliqued together with all of its parents in at least one clique node in the junction tree. After performing one pass of the junction tree algorithm, all evidence will be propagated over the junction tree and the clique nodes will contain the joint distribution of the nodes in the clique. The joint distribution of interest can then be acquired by summing out all unrelated nodes. The probabilities of the joint distribution can then be added to the appropriate counts.

The convergence of the algorithm is usually determined by defining a minimum likelihood increase. After every iteration the current likelihood of the model is compared with the likelihood of the previous state. If the difference between the likelihood is smaller than the predefined value, the algorithm terminates.

2.3.2 Structure Learning

We have described how we can learn the parameters of a Bayesian network when we already have the structure available to us. If we have a dataset available to us, we may also construct the network structure using the data. There are two main approaches to Bayesian network structure learning: 1) constraint-based search, and 2) methods using the search-and-score methodology. I discuss the two approaches in the following subsections.

2.3.2.1 Constraint Based Search Constraint based learning algorithms typically have two phases: 1) a (conditional) independence test phase and 2) an edge orientation phase. The prime example of constraint-based algorithms is the PC algorithm [Spirtes et al., 1993]. The PC algorithm starts out with a completely connected, undirected graph. It then proceeds to test pairwise (conditional) independence among all pairs of variables, i.e., for each edge in the graph. For every pair of variables where the test concludes independence, the edge is removed and the conditioning set for which the independence holds is recorded. Edges are typically tested for independence more than once. The independence procedure is run with an iteratively increasing size of the conditioning variable set. Conditional independence

among two variables is tested for every possible conditioning set. These sets are constructed from the set of adjacent variables of the two variables under consideration.

The typical test of independence for discrete variables is used for determining independence. Several variations of calculating the chi-square statistic and the degrees of freedom are utilized. The basic principle is that we assume as null hypothesis that the two variables are independent given conditioning set $\mathbf{S}$, where $\mathbf{S}$ can be empty. The most typical statistics used for the test are χ^2 and G^2 [Spirtes et al., 1993], which have minor differences. The χ^2 statistic is calculated by creating a contingency table for a variable, comparing the actually observed counts for each variable state with the hypothesized counts if the two variables would be independent. The hypothesized counts are calculated using the marginal counts for each of the variables. The χ^2 statistics can then be calculated by summing the squared differences between actual and hypothesized counts for each combination of variable states.

$$\chi^2 = \sum_{i=1,j=1}^{N_{ij}} \frac{(c_{ij} - \hat{c}_{ij})^2}{\hat{c}_{ij}} \,. \tag{2.17}$$

A χ^2 distribution can then be used to determine if we can reject the null hypothesis. The number of degrees of freedom, a required parameter for the χ^2 distribution is determined by the number of 0s we encounter in the contingency table. Different strategies are used for the calculation of the degrees of freedom. One simple approach [Spirtes et al., 1993] counts the number 0s in the table, another more complicated approach [Fienberg and Holland, 1970, Fienberg, 2007] counts the number of 0s in a row or column making sure not to count any 0 more than once.

Having discovered the skeleton of the Bayesian network, the next phase of the PC algorithm orients the edges. First, it searches for *v-structures*. V-structures are formed between triplets of nodes where we have edges between nodes A and B, and nodes C and B. A v-structure is created by orienting the edges so that we have an arc from A to B and an arc from C to B, if there exists a conditioning set $\mathbf{S}$ for which A is (conditionally) independent of B and C is not present in $\mathbf{S}$, hereby forming a v-shape.

Once all v-structures have been discovered in the skeleton, the PC algorithm iteratively checks certain node/edge/arc patterns using two rules. If one of the rules matches, it is executed and the network structure is updated[2]:

- If there is an arc from A to B, there is an edge between B and C, there is no edge between A and C, replace the edge between B and C with an arc if this does not create a new v-structure.
- If there is a directed path from A to B, meaning that there is a sequence of arcs starting at A and ending at B, and an edge between A and B, then replace the edge by an arc.

Finally, when it is no longer possible to apply any of the rules, the algorithm returns the current network structure. The algorithm might not return a Bayesian network, but a PDAG (partially directed acyclic graph), also known as a pattern. Although the PC algorithm itself does not specify any more steps to orient the remaining edges so the resulting network is a proper Bayesian network, usually the remaining edges will be oriented properly so that the end result will be a BN. Random algorithms are used to orient the remaining algorithms, but other algorithms exist that aim to orient the remaining edges such that the resulting BN is consistent with the pattern created by the PC Algorithm [Dor and Tarsi, 1992].

2.3.2.2 Search and Score The basic principle of search and score algorithms is that we define a state space, a set of operations used for transitioning between states, and a scoring function for ranking states. In the context of Bayesian networks, the search space consists of all possible directed acyclic graph (DAG) structures that can be constructed with the set of available nodes. Every state represents a proper Bayesian network structure. We define 3 state transition operators: 1) Add an arc between two nodes, 2) Remove an arc between two nodes, and 3) Invert an arc between two nodes.

The final element is the score function. Commonly, score functions are used that are based on a Bayesian principle. We define a distribution over all possible network structures and to score a structure we calculate the likelihood of a structure given the available data.

$$P\left(S\,|\mathbf{D}\right) = \frac{P\left(\mathbf{D}\,|S\right)P\left(S\right)}{P\left(\mathbf{D}\right)}. \tag{2.18}$$

[2]The rules described here and applied in the PC algorithm implementations used in this dissertation are based on the description in [Spirtes et al., 2000]

The two main components of interest of the structure likelihood are the structure prior $P(S)$ and the likelihood of the data given the specific structure $P(\mathbf{D}\,|S)$. When comparing different structures we can disregard the prior distribution of the data $P(\mathbf{D})$, since it acts as a normalizing factor and it does not depend on any specific structure S. A common choice for the structure prior is to not apply any extra weight to any of the structures, effectively defining a uniform distribution. The likelihood of the data is commonly referred to as the *marginal likelihood*. It cannot be computed directly, as to calculate the data likelihood we need a completely specified Bayesian network with a structure and a set of parameters. Since we should not pick a specific set of parameters the solution is to integrate over all possible parameter distributions, marginalizing the parameters out of the likelihood distribution. This is not a computationally feasible approach and the usual approach is to make certain assumptions to simplify the computation of the marginal likelihood [Cooper and Herskovits, 1992, Heckerman, 1995]:

- The data $\mathbf{D}$ is discrete.
- The data $\mathbf{D}$ is faithful to a Bayesian Network, i.e. there are no independences in the data that cannot be represented by a BN.
- The samples in $\mathbf{D}$ are i.i.d.
- $\mathbf{D}$ is complete, no missing values are allowed.
- A uniform prior distribution is assumed for the parameters of every BN in the search space.
- For all possible parent combinations for a variable X_i, the conditional probability distribution parameters for each of the parent combinations are independent.
- Parameter densities for the conditional probability distributions of any pair of nodes in the BN are independent.

With these assumptions, the calculation of $P(\mathbf{D}\,|S)$ can be reduced to a counting problem. Cooper and Herskovits defined the following K2 metric:

$$P(\mathbf{D}\,|S) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} (N_{ijk})!\ , \tag{2.19}$$

where q_i corresponds to the total number of parents state combinations of node i and r_i equals the number of states of node i. N_{ijk} counts the number of times node i has value k and its parents have been assigned the value j (which breaks down in a specific state assignment for each parent node). Are more general version was defined by Heckerman et al. [1995], where a Dirichlet prior distribution is used for the BN parameters, known as the BD score,

$$P(\mathbf{D}\,|S) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma\left(N'_{ij}\right)}{\Gamma\left(N_{ij} + N'_{ij}\right)} \prod_{k=1}^{r_i} \frac{\Gamma\left(N_{ijk} + N'_{ijk}\right)}{\Gamma\left(N'_{ijk}\right)} \,. \tag{2.20}$$

The BD score function typically is not used in practice. The number of hyper parameters N'_{ijk} that need to be defined can become quite large if the number of variables and/or the number of variables states increases. A commonly used score function, a special case of the BD score, sidesteps this issue by assuming a uniform structure prior [Buntine, 1991]. The BDeu score requires only one parameter, N', which represents a measure of confidence in the initial/current network structure.

$$P(\mathbf{D}\,|S) = \prod_{i=1}^{n} \prod_{j=1}^{q_i} \frac{\Gamma\left(\frac{N'}{q_i}\right)}{\Gamma\left(N_{ij} + \frac{N'}{q_i}\right)} \prod_{k=1}^{r_i} \frac{\Gamma\left(N_{ijk} + \frac{N'}{r_i q_i}\right)}{\Gamma\left(\frac{N'}{r_i q_i}\right)} \,. \tag{2.21}$$

After choosing an appropriate score function we have all elements to search for the best possible Bayesian network structure in the search space. However, it is not feasible to search through the complete search space to find the optimal network structure, as the size of the search space increases super-exponentially with the number of nodes. A common approach is to randomly generate a starting structure and search from this point. We can use a greedy hill-climbing algorithm to search for the best scoring network structure in the neighborhood of the starting point. Every time we move to the next state in the search space we pick the state that improves the score the most. This process is repeated until we reach the state that has the highest structure score or until we reach a state that is a local maximum. Typically the score function has many local maxima. The effect of this is that the greedy hill-climbing algorithm can get stuck. To find a better structure, several different strategies exist to break the deadlock, among them are:

- Random Restarts: when the algorithm gets stuck, randomly start from another point in the state space. This is repeated n times and the best structure after n searches is returned.
- Tabu Search: the last n states the algorithm has visited are kept and the algorithm is prohibited from revisiting these states again. If the algorithm gets stuck, a limited number of steps that reduce the structure score are allowed. If the algorithm does not find a state that improves on the last score within the step limit, the algorithm terminates and returns the best structure.

2.4 DISTRIBUTED COMPUTING FRAMEWORKS

With the availability of large multiprocessor supercomputers, work on parallelization of traditional sequential algorithms became much more widespread. Adding to that the availability of computational clusters consisting of relative inexpensive machines, the ability to process large amounts of data, unavailable before due to the limits single computers, has become available to a large audience. This has spurred the efforts to standardize approaches to algorithm parallelization and interprocessor/intercomputer communication. I discuss two popular paradigms for parallel computation: the Message Passing Interface standard, and the MapReduce framework.

2.4.1 Message Passing Interface

The Message Passing Interface (MPI) standard [Snir et al., 1995] is commonly used for implementing parallel algorithms on a diverse array of parallel computation platforms. Examples of these platforms are large, singular machines with many processors and shared memory, or a large array of separate machines collected in a cluster. The MPI standard defines a protocol for passing messages among processing units, which could be processors or complete computers. Many library implementations have been developed that follow the standard. MPI is available for all the popular platforms and in many popular programming languages.

Processing unit can communicate directly through a point-to-point mechanism, but users of the library can define arbitrary logical topologies over the processing units.

The low level technicalities of interprocess communication are abstracted away by the library but the user still retains a large amount of control over how to utilize available resources. Using the library, different computational paradigms can be implemented, a popular one being the master/worker paradigm, where a single controller is responsible for the algorithm workflow, assigning work to the other processing units.

2.4.2 MapReduce

MapReduce is a parallel computation framework developed at Google [Dean and Ghemawat, 2004]. It differs from most parallel computation approaches by very specifically defining two types of computational functions. These functions are called mappers and reducers. Mappers take an input, perform some sort of calculation and then output one or more key-value pairs. The outputted pairs are collected, sorted, and then assigned to reducers. Reducers are aggregate functions. They are assigned collections key-value pairs, where the keys all have the same value. The reducer iterates over all values it has been assigned and typically calculates some sort of aggregate result, which then is emitted as a new key-value pair, where the original key from the inputs is used as the key in the new pair.

One of the main benefits of the MapReduce framework is that it abstracts away just about all details from the parallelization process. The main responsibility of any MapReduce user is to define the mapper and reducer functions. All other parallelization details such as load balancing, fault tolerance, distributed data storage are taken care of in the background. One popular example of an open source implementation of a MapReduce framework is Hadoop [White, 2012], which was developed mostly at Yahoo!

The MapReduce framework excels at processing data at an enormous scale. It has been used for tasks such search engine indexing and log file processing. Companies such as Google, Yahoo! and Facebook have many petabytes of data that needs to be processed. Facebook currently maintains the largest Hadoop cluster that contains over 100 petabytes of data.

3.0 COMPARING STRUCTURE REPRESENTATIONS AND DISTANCE MEASURES

In this chapter, I report the results of a series of experiments that I have performed with the goal of empirical comparison of various measures of structural distance.

3.1 INTRODUCTION

An important outgrowth of the work on Bayesian networks is the field of causal discovery, focusing on retrieving from data the causal structure of the system that has generated it. While originating somewhere at the intersection of computer science, statistics, and philosophy [Cooper and Herskovits, 1992, Spirtes et al., 1993, Pearl and Verma, 1991], this area has had a profound impact on science in general. Many philosophers of science consider that true understanding, one of the main foci of science, means causal understanding [Salmon, 1984]. Without understanding the causal structure of a system, a human decision maker or an artificially intelligent agent will not be able to predict the effect of those of their actions that impact the system. For example, all individual and policy decisions concerning smoking tobacco hinge on the common causal knowledge that inhaling tobacco smoke has a negative impact on health.

Causal discovery is closely related to the field of learning graphical models. While both fields use similar algorithms, there is a subtle difference in that the structure and, in particular the directions of edges, is of utmost importance in causal discovery and may be less essential for other applications. For example, in a machine learning application like classification, it is immaterial what the exact DAG structure looks like, as long as the DAG

performs well as a classifier. The exact topology of the DAG will merely have an impact on the number of necessary parameters and the quality of predictions. However, when the goal is to establish the causal relations among the variables in the data, it is the structure that counts.

One approach to evaluating the performance of a structure learning algorithm is to start with an existing causal graph $\mathcal{G}_G$, generate a dataset from the joint probability distribution that it represents, and subsequently use the algorithm to retrieve the original structure. The main advantage of this approach is that the resulting graph $\mathcal{G}_L$ generated by the structure learning algorithm can be compared with the structure of the original graph $\mathcal{G}_G$, the gold standard that generated the dataset. A critical element of this approach is a measure of structural distance between $\mathcal{G}_G$ and $\mathcal{G}_L$: the smaller this distance, the better the algorithm was in retrieving the original structure.

There are several measures of structural distance that can be found in the literature. None of these has been widely accepted as the primary measure. This is somewhat surprising, as there are sound theoretical grounds for preferring one group of the measures over another. Because there typically exist several DAGs that can represent the same joint probability distribution (often lumped into so-called equivalence classes), they cannot be distinguished from one another based purely on the data. Hence, comparing DAGs directly, with disregard for the theoretical impossibility to differentiate between them, does not seem to be a sound approach. Still, some researchers focus on comparing DAGs, others on comparing equivalence classes of DAGs.

To my knowledge[1], no systematic comparison of different measures and representations has ever been performed and little is known how the different measures behave with relation to one another in practice. A researcher testing a structure learning algorithm is thus left with the uninformed choice of measure and structure representation.

In this chapter, I report the results of a series of experiments that I have performed with the goal of empirical comparison of various measures of structural distance. A surprising finding, originating from my experiments, in stride with theoretical expectation, is that there seem to be no significant differences between comparing DAGs and comparing their equiva-

[1]Parts of Chapter 3 have been previously published in [de Jongh and Druzdzel, 2009]

lence classes. I have found no apparent systematic effects, outside of individual differences in the graphs and data, that would favor one measure over another. The different measures are correlated with one another but there are no sufficiently strong grounds for picking one and disregarding others. A recommendation that originates from my work is that researchers use several measures when reporting the accuracy of their algorithms.

3.2 STRUCTURAL DISTANCE MEASURES

There are two fundamental approaches to comparing a learned DAG structure $\mathcal{G}_L$ to a gold standard structure $\mathcal{G}_G$: (1) comparing structures $\mathcal{G}_G$ and $\mathcal{G}_L$ directly, and (2) comparing equivalence classes $E(\mathcal{G}_G)$ and $E(\mathcal{G}_L)$.

The former is more widespread than the latter, even though there are sound theoretical reasons for preferring equivalence classes comparisons over DAG comparisons. I review these two approaches in Sections 3.2.1 and 3.2.2 respectively.

3.2.1 DAG Measures

There are many examples of measures of structural distance that rely on comparing DAGs. Cooper and Herskovits [1992], for instance, compare the performance of their K2 algorithm for different sample sizes and look at added edges and missing edges, i.e., edges in the discovered graph that either were not present in the original or should have been present in the discovered graph but were missing respectively. A similar approach is taken by Monti and Cooper [1997]. Heckerman et al. [1995] mention a structural difference metric (symmetric difference of parents of a node in gold standard and learned structure). Colace et al. [2004] describe multiple metrics and introduce two normalized versions of DAG metrics. Cruz-Ramírez et al. [2006] evaluate the performance of the Bayesian Information Criterion (BIC) and the Minimum Description Length (MDL) principle as model selection metrics. In one part of their evaluation, they compare learned DAG structures against gold standard networks.

3.2.2 Pattern Measures

In an example involving their PC algorithm, Spirtes et al. [1993] mention metrics similar to extra and missing edges: they call them edge commissions and omissions, but they base their metrics on patterns, not on DAGs. Abellán et al. [2006] and Cano et al. [2008] use similar metrics for DAGs in their evaluation of variations on the PC algorithm, They calculate their metrics before the edge orientation phase of the PC algorithm. To evaluate their hybrid algorithm, Tsamardinos et al. [2006] compared the output of their algorithm against a gold standard network. They specifically compare equivalence class structures and not DAGs. They define a metric called the *Structural Hamming Distance* (SHD), which is similar to the metric proposed by Acid and de Campos [2003]. They compute the SHD between two directed graphs after first converting them into patterns by using the approach of Chickering [1995]. Perrier et al. [2008] propose a small modification to the SHD metric, assigning a smaller penalty to incorrect orientation of edges.

3.2.3 Differences

If we compare the first two DAG structures in Figures 2.1(a) and 2.1(b) directly to each other, focusing on differences in edge direction, we will find a distance of 2. This is because of the two edges in Figure 2.1(b) that are reversed with respect to their equivalents in Figure 2.1(a). If we compare the pattern structures derived from both DAGs, we will obtain a distance of 0: Because both DAGs belong to the same equivalence class, they will have the same pattern structure.

Similarly, if we compare the DAG structures in Figures 2.3(a) and 2.3(c) directly to each other, focusing on differences in edge direction, we will find a distance of 2. This is because of the two edges in Figure 2.3(c) that are reversed. If we compare the pattern structures derived from both DAGs, we will obtain a distance of 0: Since both DAGs belong to the same equivalence class, they will have the same pattern structure.

These simple examples show that discrepancies are possible between DAG and pattern scores. If Figure 2.1(a) is the result of a structure learning algorithm, and another algorithm produces the DAG of 2.1(b), can we say that this second algorithm performs better? Because

of statistical indistinguishability (Section 2.1), the difference may be incidental rather than stemming from the data. When the patterns are compared, both graphs yield the distance 0 from the pattern and have to be considered equivalent.

3.3 EMPIRICAL EVALUATION

In this section, I report the results of a series of experiments that focus on empirical comparison of various measures of structural distance.

3.3.1 Experiment 3.1: Basic Quantitative Evaluation of the Distance Measures

In my first experiment, reported in [de Jongh and Druzdzel, 2009], I compared various distance measures for DAGs. While my study is independent of the choice of a structure discovery algorithm, I used a Bayesian search algorithm, Greedy Thick Thinning [Heckerman, 1995] (GTT). GTT returns DAGs, although a DAG can be converted into the pattern representing its equivalence class [Chickering, 1995]. This is important for my evaluation, since I want to compare measures used for both DAGs and patterns.

3.3.1.1 Methodology I chose four different existing BNs of varying sizes to use as gold standard DAGs: Asia [Lauritzen and Spiegelhalter, 1988], Alarm [Beinlich et al., 1989], Hailfinder [Abramson et al., 1996], and HEPAR [Onisko, 2003]. For every DAG, I generated datasets of three different sizes: 1,000, 5,000, and 10,000 records. I generated 100 datasets of each sample size to be able to acquire useful statistics for the different measures under study. For every combination of DAG and sample size, I compared the result of the learning algorithm with the gold standard twice, focusing on: (1) the DAG structures, and (2) their equivalence classes. I examined the following metrics and measures:

1. **Missing Edges**: counts edges that are present in the original structure but are missing in the learned structure.

2. **Extra Edges**: counts edges that are found in the learned structure but are not present in the original structure.
3. **Correct Edges**: counts edges, regardless of their orientation, that are present both in the original structure and the learned structure.
4. **Correct Type Edges**: counts edges, taking into account their orientation, that are present both in the original structure and the learned structure.
5. **Correct Edge Direction**: counts directed edges in the learned structure that are oriented correctly.
6. **Incorrect Edge Direction**: counts directed edges in the learned structure that are oriented incorrectly.
7. **Topology**: Measure is a normalized, weighted combination of measures 1,2, and 3 [Colace et al., 2004].
8. **Global**: Measure is similar to the topology metric, combining measures 1,2,4, and 5 [Colace et al., 2004].
9. **Hamming Distance**: Describes the number of changes that have to be made to a DAG for it to turn into the one that it is being compared with. It is the sum of measures 1, 2, and 6. I have found multiple mentions of the metric in the literature: [Acid and de Campos, 2003, Tsamardinos et al., 2006, Perrier et al., 2008].

I calculated the mean, variance, standard deviation, minimum value, and maximum value. I plotted several series of averages to look for trends. Examples of examined series are: comparing metric values for DAG or pattern while varying dataset size, comparing the DAG and pattern metric for one network while varying dataset size, and comparing the DAG and pattern metric for one dataset size while varying the network.

3.3.1.2 Results

I do not show the results for all the metrics that I examined but will instead summarize the results and illustrate my most important findings. I found that the distances reported by the metrics on either DAG structures or patterns are similar. This can be seen in Figure 3.1, which shows for each of the DAGs the Hamming distance metric as a function of the size of the dataset.

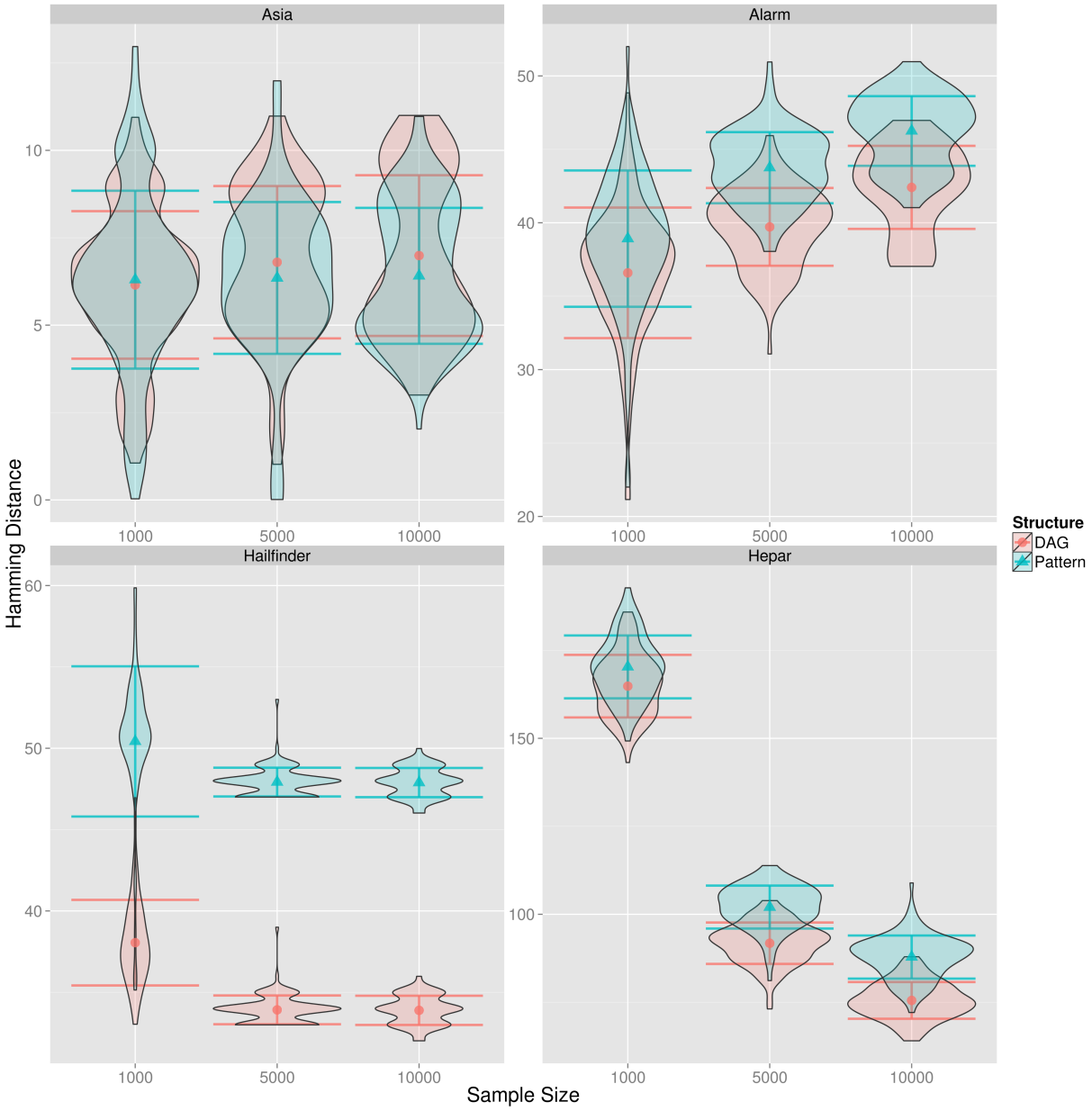


Figure 3.1: Similarity of values of the Hamming distance metric

Second, I found that there are situations where it does matter whether the distance is measured between DAG or pattern structures. I show examples of this in Figure 3.2, which contains plots of normalized values of two measures, correct edge directions and incorrect edge directions (Figure 3.2). The values of the two measures are normalized by the total number of edges in the original DAG (incorrect edge direction), and the total number of

directed edges in the original DAG (correct edge direction). Depending on the DAG that I used to generate the training data, the structures that the learning algorithm retrieved had quite different average values for the counts of correct and incorrect edge directions for the DAG and the pattern cases. This is apparent especially in the Hailfinder network. On the average, both show that the learning algorithm retrieved about 70% of the directed edges. But there is a large discrepancy between the counts of the incorrect edge directions.

I have an explanation for these differences. The correct edge direction measure examines only directed edges and the incorrect edge direction measure examines directed and undirected edges. The correct interpretation of the graphs is that in both structures 70% of all the directed edges are retrieved. But for the DAG, this means 70% of all edges and for the pattern this means 70% of the directed edges. These edges are either parts of, or originate from v-structures. Since the same structure is compared under two circumstances, we can say that the directed edges included in the pattern are a subset of the directed edges that are present in the DAG. Since a DAG will usually, and certainly for the Hailfinder network, have more directed edges than the corresponding pattern representation, some of these will be statistically indistinguishable. Some edges in the DAG may have been correctly identified by coincidence. This is most visible in the incorrect edge direction, where the lower pattern score could be explained by the fact that patterns have three different possible edges and DAGs only have two. If two randomly generated graphs with exactly the same skeleton are compared, DAGs would have a 50% chance for each individual edge to be correct. For patterns, this would only be 33%, since there are three possibilities: directed edge from a to b, directed edge from b to a, or an undirected edge. More errors are simply to be expected.

This is counterintuitive from the theoretical point of view. An equivalence class can contain many DAGs, and, because in patterns edges can be undirected, there should be fewer errors due to statistical indistinguishability. But even when all correct edges in a pattern are counted, directed and undirected, there is still a huge gap between DAGs and patterns. I found for the Hailfinder network that, on average, about 52% of all edges in the original DAG are retrieved correctly versus 72% in the DAGs. The data suggests that the learned DAG structure appears closer to the original DAGs than the respective equivalence classes that they belong to.

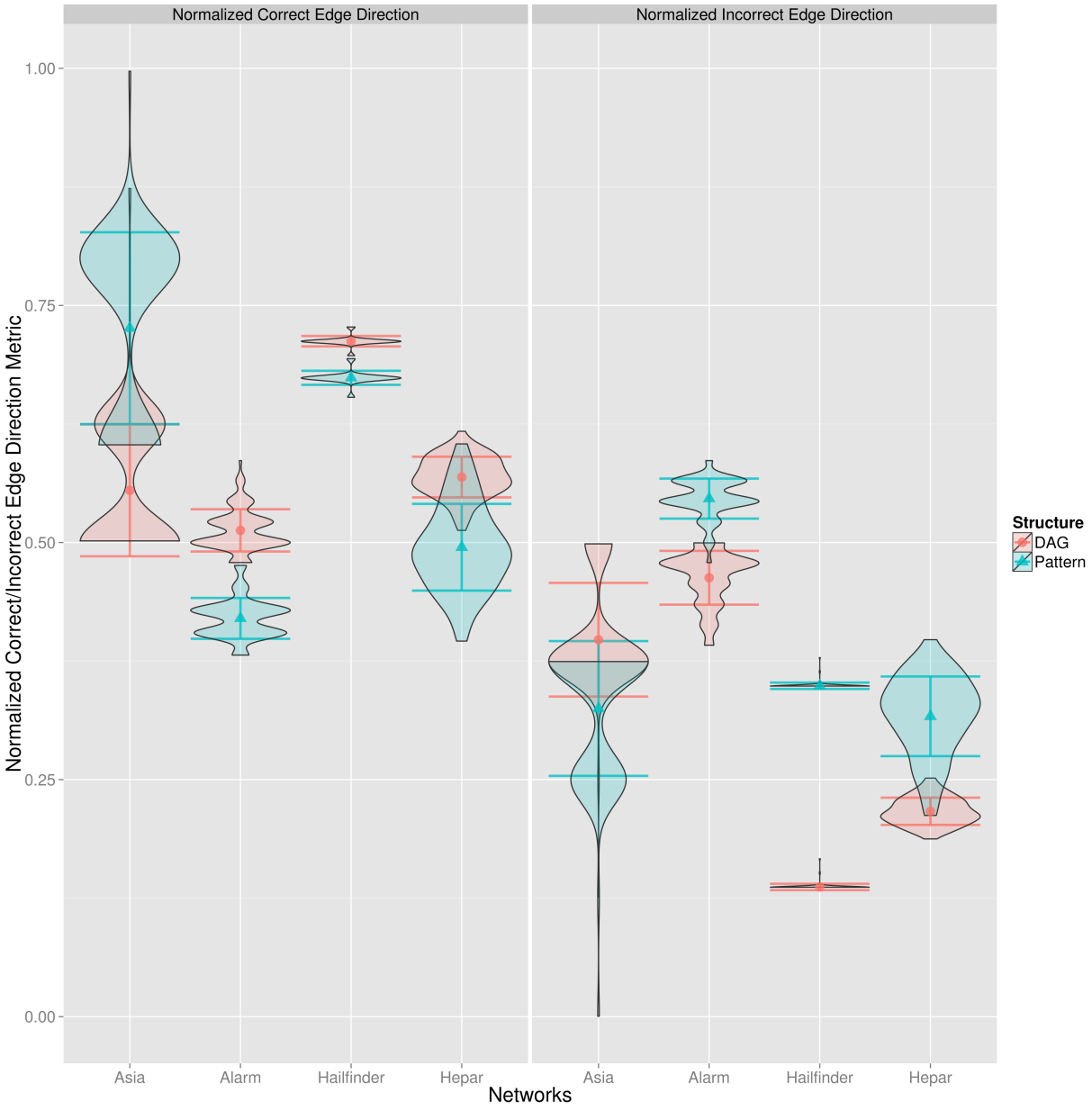


Figure 3.2: Normalized values of measures for DAGs learned with 10,000 samples

I observed that sometimes, when more data were available for learning, the distance to the original DAG would increase. This is clearly visible in the Alarm network in Figure 3.1, where the Hamming distance increased when more data were available for learning. Further investigation showed that factors contributing to the increase of the Hamming distance were extra edges and incorrect edge directions.

Finally, I want to comment on the Hamming distance metric. It is a proper metric, and it is a useful indicator of structural distance. But it is unwise to rely solely on this metric, or any single metric, in general. As an example, consider Figure 3.3. Here, the Hamming metric shows a clear trend. As more data are available, the learned DAG seems to better resemble the original DAG. However, when I examine the individual components of the Hamming distance metric, I see that when more data becomes available, the skeleton of the structure improves, but the number of incorrectly oriented edges increases. Since the increase of incorrect edges is smaller than the combined decrease of missing and extra edges, this information is lost when considering only the Hamming distance. Specifically, when evaluating algorithms for causal discovery, such information might be vital and relying only on the Hamming distance as a measure of distance may be misleading.

3.3.2 Experiment 3.2: Impact of *v*-Structures on the Distance Measures

From the theoretical perspective, a major factor in edge orientation is presence of v-structures (see Section 2.1). My second experiment focused on disambiguating the role that presence of v-structures has in the difference between distance measures based on DAGs and patterns.

To be able to control the number of v-structures in an DAG, I randomly generated DAGs that would have a lower and an upper bound on the percentage of their v-structures.

3.3.2.1 Methodology I created a DAG generator [Ide and Cozman, 2002, Melancon et al., 2000] that makes use of Markov Chain Monte Carlo sampling to randomly walk through the state space of all possible DAG structures and finds a candidate structure. In addition to constraining the number of v-structures the DAG has, I enforced the number of edges to be $O(n)$, where n is the number of nodes in the DAG. Real DAGs are typically sparse and their number of edges typically stays linear in their number of nodes.

For the experiment I created the following setup:

- Network sizes: {10, 20, 40, 80} nodes.
- Two approaches to determine v-structure percentage

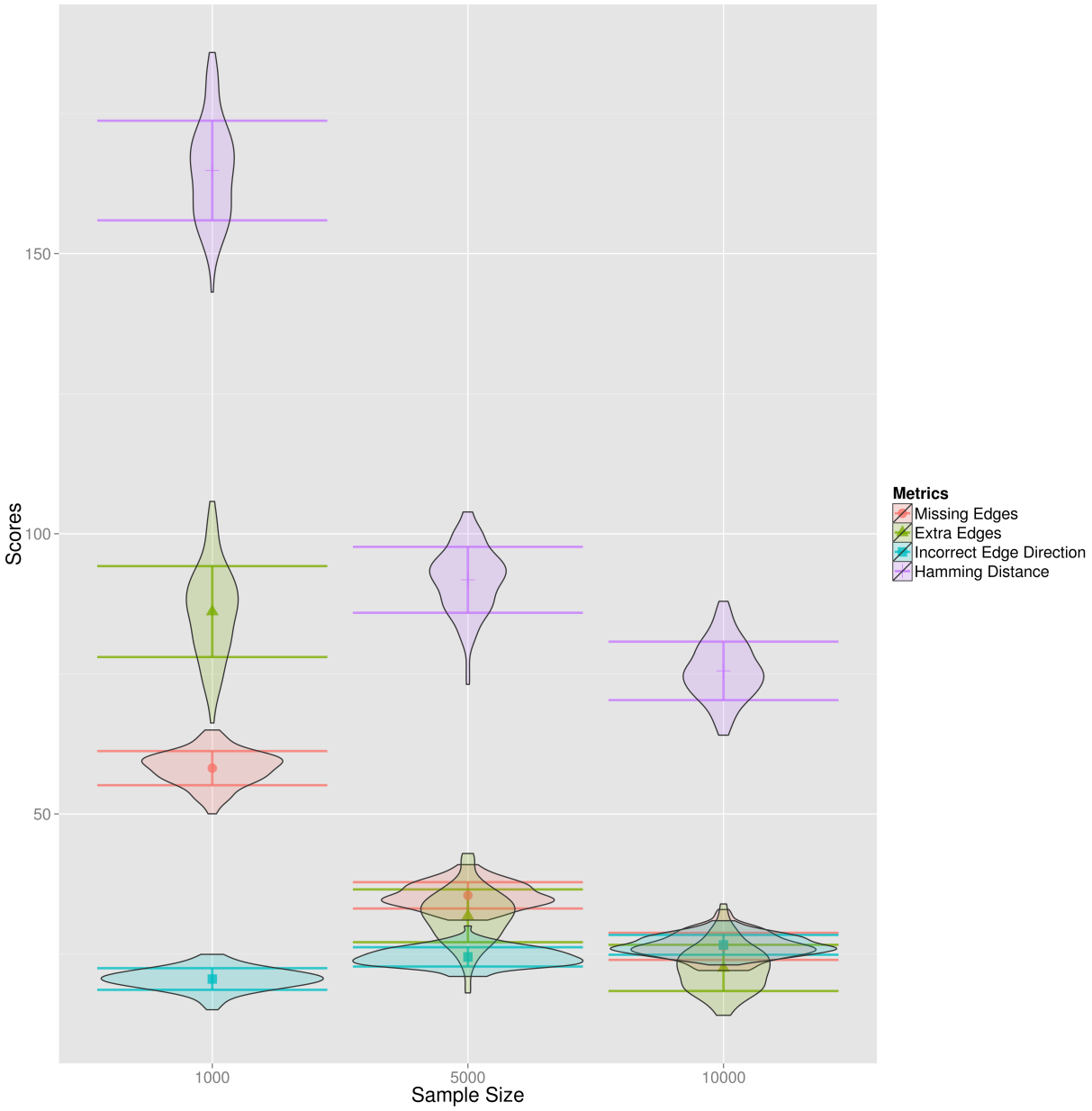


Figure 3.3: Hamming metric and its components on the Hepar network

  - The ratio of compelled edges [Chickering, 1995] to total edges.
  - The ratio of nodes with 2 or more parents to the total nodes.
- Percentage of v-structures: {0-25%, 25-50%, 50-75%, 75-100%}.
- For each network size a baseline network was generated without any restriction on v-structures

- Sample sizes: {1,000, 5,000, 10,000} samples.
- Learning Algorithms used:
  - Greedy Thick Thinning [Heckerman, 1995] (GTT)
  - Bayesian Search [Cooper and Herskovits, 1992] (BS)
  - PC [Spirtes et al., 1993]

In total I generated nine DAGs for each network size, and from each generated DAG I generated 100 datasets for each dataset size. I presented these datasets to both algorithms and compared the learned structures to the original DAGs using the Hamming Distance metric.

I used two different approaches to generate random DAGs with a certain percentage of v-structures. While both have their advantages and disadvantages, neither of them exactly represents the percentage of v-structures in a DAG. This is not a trivial thing to define. When a DAG has no v-structures, it is easy and correct to say that the percentage of v-structures in the DAG is zero. It is more difficult to define 100%. It is possible to construct a DAG in which every triple of nodes forms a v-structure, but things get more difficult when we add nodes with more than two parents. The main difficulty is to determine the percentage if it is neither 0 or 100%. We can easily count the number of v-structures in a DAG, but to get a percentage, we have to have a maximum value of some sort to normalize the number. This may be related to the number of edges, the number of nodes, or the topology of the DAG.

I chose to work with the two approaches mentioned above because they are easy to calculate and are reasonable approximations of the percentage of v-structures. The compelled edges approach counts the number of edges in the equivalence class of a DAG and then divides this by the total number of edges in the DAG. When an DAG has more v-structures, more edges will be "compelled" in its equivalence class structure. It may overestimate the percentage somewhat due to some edges being considered "compelled" but not being part of a v-structure. The "2+ parents" approach simply checks if a node has 2 or more parents and will count this node if this is indeed the case. The total number of nodes with 2 or more parents is then divided by the total number of nodes to get an estimate. This measure

has a tendency to underestimate the percentage of v-structures. The number of v-structures involving a node and n parents can be described as $\frac{n(n-1)}{2}$, and will increase quadratically.

3.3.2.2 Results I expected that recovering the DAG structure from a dataset generated by a DAG that has many v-structures would be easier. Instead, I found that with an increase of the percentage of v-structures in the DAGs, the Hamming distances of both measures increased, which means that the DAG structures learned from data resembled the original structures less.

Figure 3.4 shows the effect of the two different approaches of determining the percentage of v-structures in a DAG. The graphs show the average Hamming distance (over 100 DAGs learned) and the standard deviation for patterns the network structures learned by the three algorithms from of the 20 node random networks data, as a function of the percentage of v-structures in the DAG. I show results of DAGs generated using the "compelled edges" method and "2+ parents" approach.

When we examine Figure 3.4, we see that both show similar results and trends, but the networks generated by using the "2+ parents" approach were more difficult to recover from data, contrary to my theoretical expectations.

The main results of the experiment are shown in Figures 3.5 and 3.6. Figure 3.5 shows the results for the 10 and 20 node DAGs, Figure 3.6 shows the results of the 40 and 80 node DAGs.

Sample size has an influence on the results, but this is not always significant. Learning DAGs using more samples usually gives better results, but the influence of sample size on the average Hamming distance is not very large in general. To test the impact of datasets with a larger number of samples I reran the experiment with datasets of 100,000 records. I did not find significant differences with the results of the original experiment.

Besides comparing the effect of increasing the percentage of v-structures, I have also looked at what happens to the average Hamming distance when we increase the number of nodes without any restriction on v-structures. For each DAG size, I had generated a DAG that was not limited to a specific percentage of v-structures. Then, like in case of the other DAGs I generated 100 datasets and ran the learning and scoring procedure.

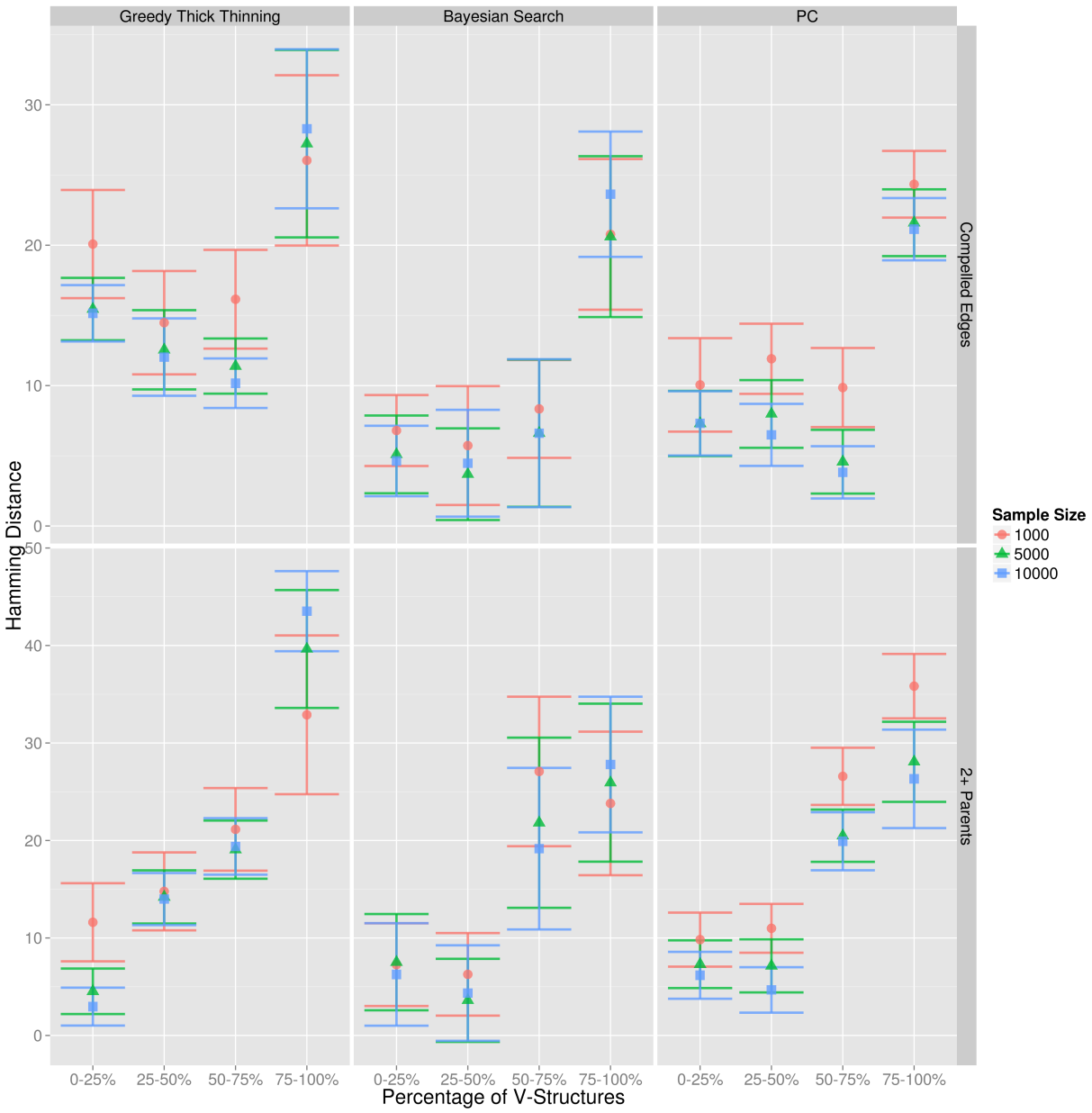


Figure 3.4: Influence of different methods for determining v-structure percentage

I show the results in Figure 3.7. The plot of the results of these baseline DAGs show that the average Hamming distance increases when the size of the DAGs increase. Both algorithms show the same trend, roughly linear increase, except for the 1,000 sample size results which seems to increase faster.

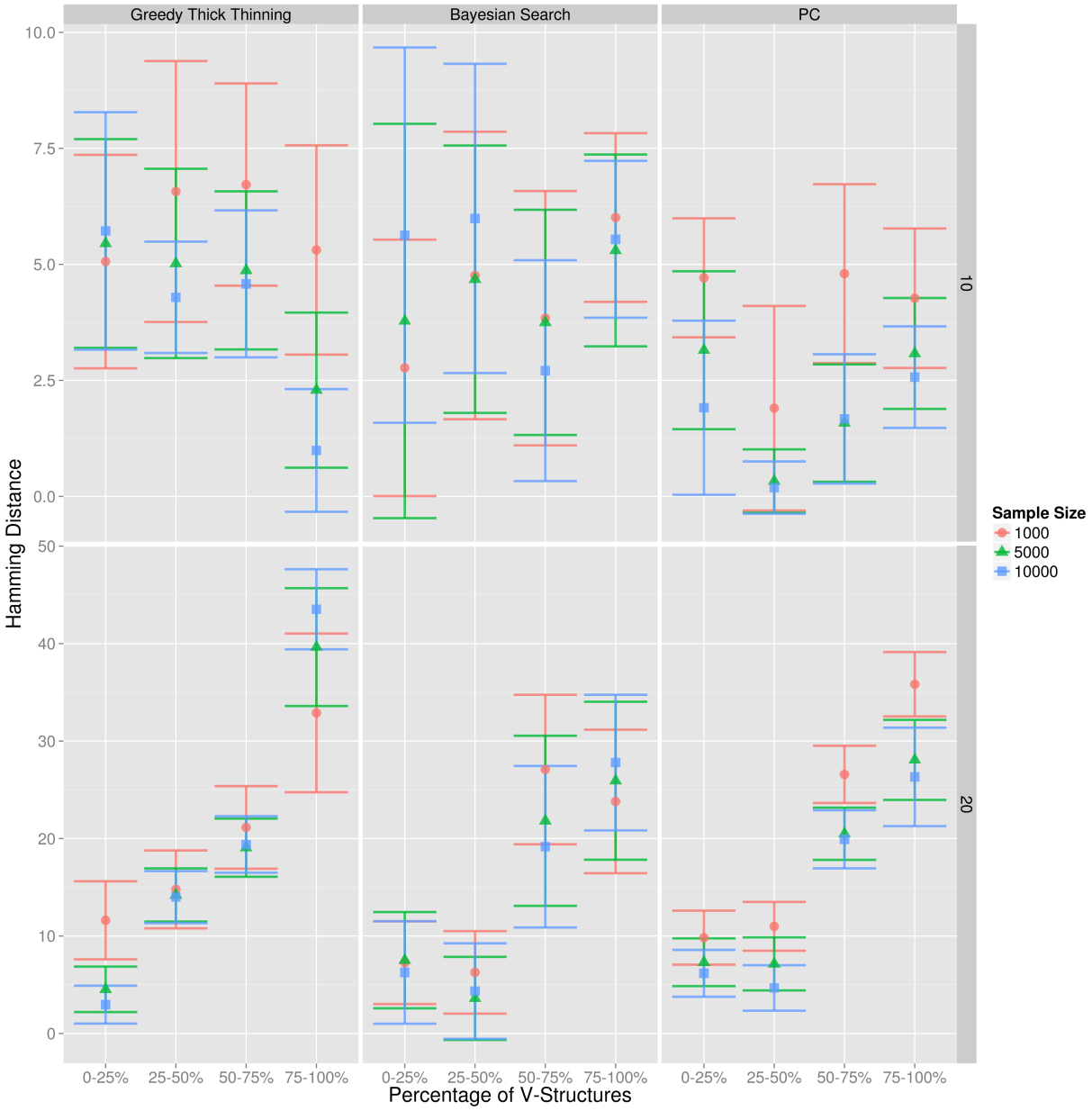


Figure 3.5: Results of Experiment 3.2: 10 and 20 nodes

3.3.3 Experiment 3.3:
Impact of the Size of Equivalence Classes on the Hamming Distance

One surprising results of Experiment 3.1 was that the Hamming distances between DAGs were smaller than the distances between their equivalence classes. One possible reason for

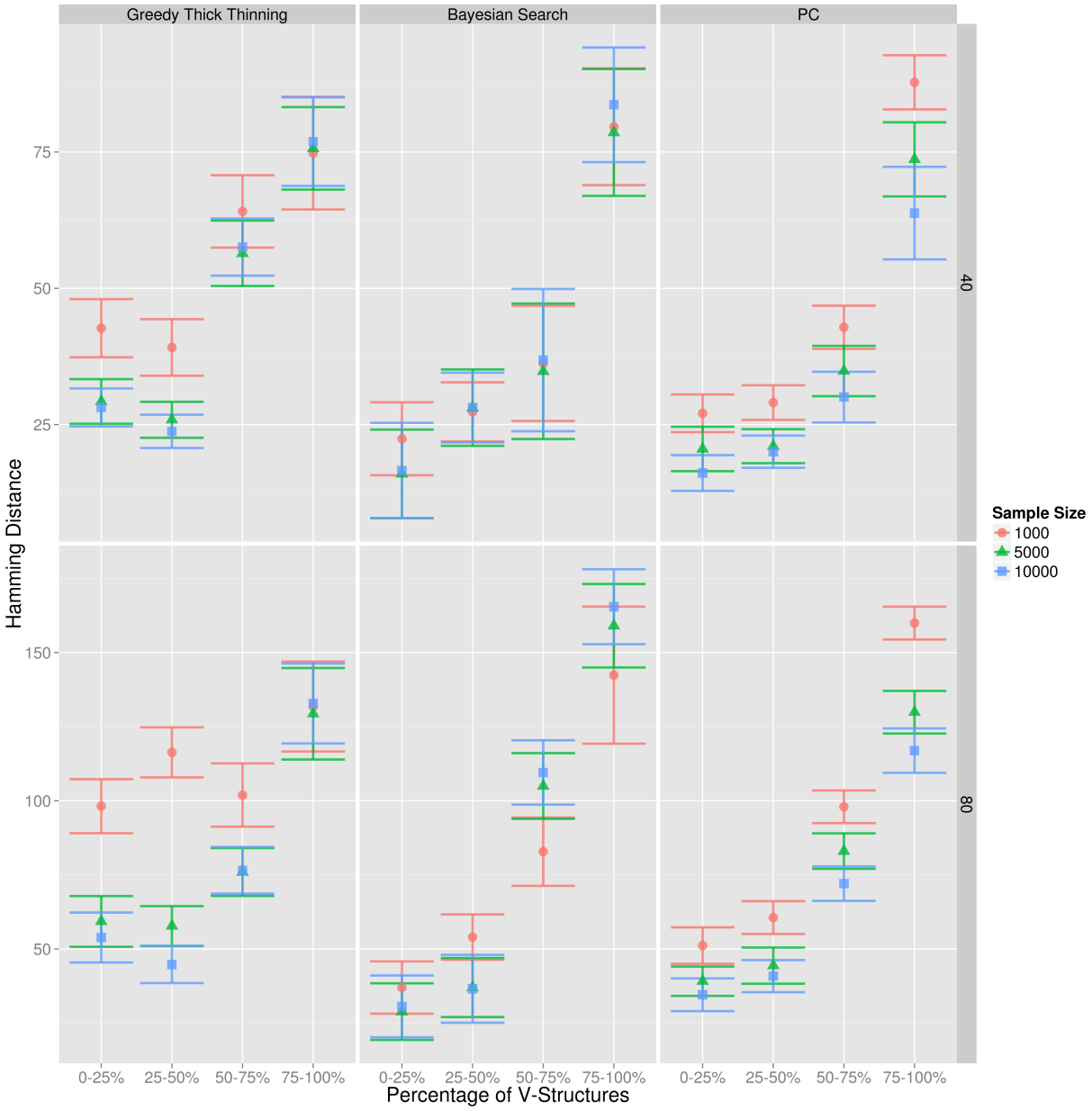


Figure 3.6: Results of Experiment 3.2: 40 and 80 nodes

this could be the size of the equivalence classes: The larger the class, the smaller the distance between instances of a graph. My next experiment focused on the sizes of the equivalence classes in Experiments 3.1 and 3.2. Using Chickering's algorithm [Chickering, 2002], I have determined the equivalence class structures of each of the DAGs of Experiment3.1 and 3.2. I performed the same procedure for two additional networks, Win95pts and CPCSS179.

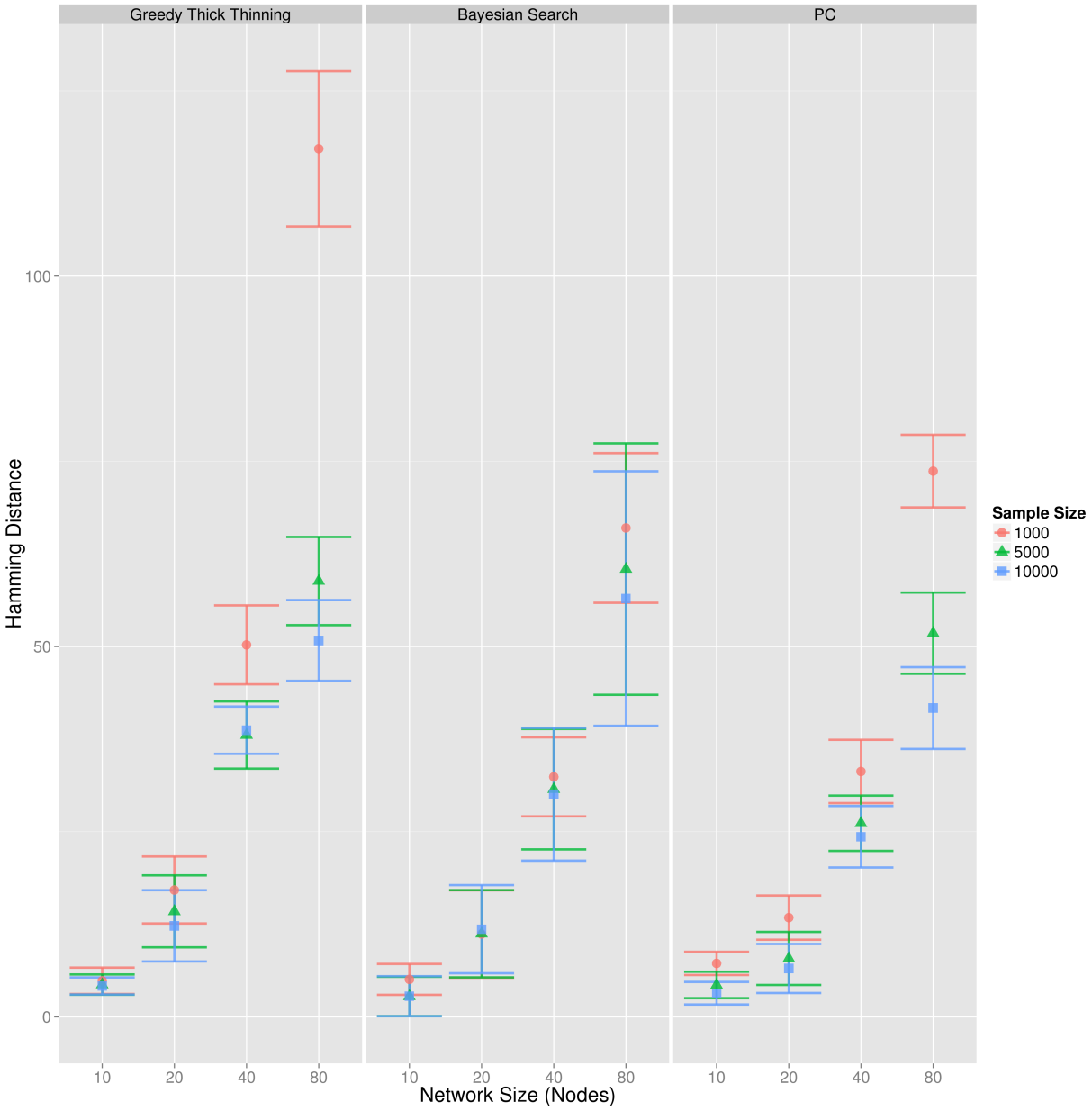


Figure 3.7: The general trend shown by the baseline DAGs

Finally, I used a brute force algorithm that generated all possible DAGs from a pattern and counted these. The edges that are part of a v-structure and a few select edges that originate from v-structure are the only edges that will be directed. All other edges in the structure are undirected. A pattern that has more v-structures should represent fewer DAGs and, hence, the equivalence class that it represents should be smaller. This should have a direct

impact on the Hamming distance. Besides counting the DAGs that resulted from orienting all undirected arcs in the pattern, I have also compared the DAGs with the structure of the DAGs used for obtaining the pattern.

3.3.3.1 Methodology I have determined the size of the equivalence classes for the following networks:

- Networks from Experiment 3.1:
  - Asia
  - Alarm
  - Hailfinder
  - Hepar
- The random DAGs from Experiment 3.2
- Two more networks from the literature:
  - Win95pts
  - CPCSS179

There are certain requirements each class member candidate has to meet. (1) The newly oriented edges may not create a cycle in the graph, and (2) no new v-structures may be created. Every structure candidate that meets all requirements is compared against the structure of the original DAG and the measurements of the distance measures are stored in a file for further processing.

3.3.3.2 Results Table 3.1 shows the results of this experiment. CE stands for Compelled Edges [Chickering, 1995], which are directed edges in patterns. VS stands for v-structures, CM (for Max CM and CM columns) stands for Class Members. Max CM contains the theoretical upper limit of the number of equivalence class members and CM contains the actual number of members found after running the brute-force procedure. CHA and CHS are the average and standard deviation of Hamming distances that were measured between the structure of the original DAG and the generated members of the equivalence class.

Figure 3.8 shows a log-log plot of the actual size of the equivalence class of each of the DAGs plotted against its theoretical maximum. The number of class members for each

Table 3.1: Results of Experiment 3.3

Network	Nodes	Edges	CE	VS	Max CM	CM	CHA	CHS
Asia	8	8	5	2	8	6	1.17	0.75
Alarm	37	46	42	24	16	16	2	1.03
Hailfinder	56	66	49	34	131,072	18	1.83	0.51
Hepar	70	123	114	100	512	96	2.83	1.07
Win95pts	76	112	100	129	4,096	640	4.7	1.52
CPCS179	179	239	230	122	512	18	2.28	0.96
rn10	10	12	8	4	16	9	2	1.22
rn10a	10	9	2	1	128	14	2.36	1.39
rn10b	10	9	4	3	32	10	2.1	1.20
rn10c	10	9	6	2	8	6	1.5	1.05
rn10d	10	20	15	9	32	10	2.1	1.20
rn10e	10	9	3	1	64	16	1.75	0.86
rn10f	10	9	7	3	4	4	1	0.82
rn10g	10	9	7	5	4	4	1	0.82
rn10h	10	9	9	4	1	1	0	-
rn20	20	25	17	11	256	48	3.17	1.42
rn20a	20	19	4	2	32,768	72	4.89	2.07
rn20b	20	19	9	5	1,024	96	4.5	1.77
rn20c	20	19	14	8	32	24	2.5	1.22
rn20d	20	33	32	19	2	2	0.5	0.71
rn20e	20	19	10	12	512	288	4.17	1.47
rn20f	20	20	15	8	32	24	2.17	1.01
rn20g	20	37	34	23	8	8	1.5	0.93
rn20h	20	48	43	33	32	16	2.5	1.37
rn40	40	51	41	23	1,024	96	4.08	1.53
rn40a	40	39	9	8	1,073,741,824	2,016	6.88	2.23
rn40b	40	39	19	14	1,048,576	11,340	7.50	2.14
rn40c	40	39	22	10	131,072	800	6.35	2.23
rn40d	40	68	54	41	16,384	432	5.11	1.68
rn40e	40	39	23	9	65,536	1,000	5.9	1.97
rn40f	40	45	35	21	1,024	96	3.92	1.56
rn40g	40	66	56	45	1,024	64	4.75	2.18
rn40h	40	120	116	106	16	5	1	0.71
rn80	80	102	92	52	1,024	768	5	1.63
rn80a	80	79	19	9	1.15292E+18	174,960	15.84	4.6
rn80b	80	79	39	30	1.09951E+12	53,747,712	16.33	3.40
rn80c	80	79	48	29	2,147,483,648	7,464,960	13.93	2.96
rn80d	80	114	101	65	8,192	864	5.5	1.74
rn80e	80	79	47	24	4,294,967,296	967,680	13.39	3.13
rn80f	80	84	59	32	33,554,432	103,680	9.27	2.29
rn80g	80	135	119	77	65,536	464	4.34	1.36
rn80h	80	207	201	195	64	12	2.17	1.11

equivalence class of the tested DAGs was much smaller than expected. The first surprise came when calculating the theoretical maximum number of class members for each DAG. Even though some DAGs had many nodes and edges, the maximum number of class members did not reflect my expectations that large DAGs would generally have larger equivalence classes. The second surprise was that the sizes of the equivalence classes were often small compared to the theoretical maximum, especially when the theoretical sizes were large. For example, an 80-node DAGs with a theoretical maximum of over a quintillion members, had only about

Table 3.2: Correlation matrix of results Experiment 3.3

	N	E	CE	VS	Max CM	CM	CHA	CHS
N	1.0000							
E	0.8947	1.0000						
CE	0.8067	0.9744	1.0000					
VS	0.6510	0.9011	0.9341	1.0000				
Max CM	0.1713	0.0633	-0.0823	-0.0903	1.0000			
CM	0.1980	0.0731	-0.0194	-0.0139	-0.0256	1.0000		
CHA	0.4517	0.1747	-0.0360	-0.0734	0.4544	0.5317	1.0000	
CHS	0.3481	0.0915	-0.1178	-0.1429	0.5859	0.3957	0.9525	1.0000

175,000 real class members. A mere fragment of the whole.

Figure 3.9 shows the actual size of the equivalence class of each of the DAGs plotted against the percentage of v-structures in the DAG. The percentage of v-structures in a graph seems to have almost no influence on the actual number of class members. I expected that having more v-structures would have a limiting effect on the number of class members.

I have examined the data and have calculated a correlation matrix for the different columns of Table 3.1. The results are shown in Table 3.2.

The most interesting and strongly correlated, variable pairs seem to be: (N, E), (N,CE), (E,CE), (E,VS), and (CE,VS). It is not surprising that the number of nodes (N) has strong influence on the number of edges (E) and the number of compelled edges (CE). In any valid DAG, there cannot be unconnected nodes. Therefore, there should always be a minimum number of edges $E = N - 1$. A similar explanation can be given for the strong correlation between N and CE. When the number of nodes increases, with the increase of the number of edges, more edges can become compelled edges. This is further reinforced by the strong correlation between E and CE.

The number of v-structures (VS) seems to be strongly correlated with both the number of edges (E) and the number of compelled edges (CE). The same line of reasoning as before can be applied here. To be able to have more v-structures, more edges must be present. Edges that are part of v-structures are by definition compelled edges, so any increase in v-structures automatically means an increase of compelled edges. The strong correlation between CE and VS supports my choice for using the ratio of compelled edges to all edges as an estimation

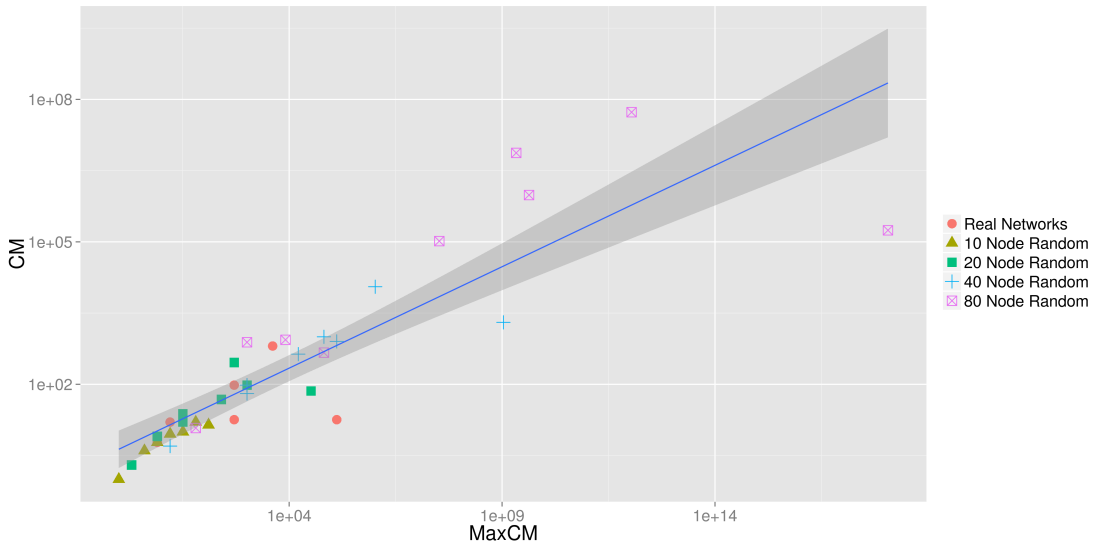


Figure 3.8: Actual size of the equivalence class as a function of the theoretical maximum class size

of the percentage of v-structures for random DAG generation. It is a reasonable assessment of structure complexity.

Finally, there seems to be a absence of any strong correlations between the real number of class members (CM) of an equivalence class and any of the other variables. One notable exception is shown in Figure 3.8, where I compare the log of the theoretical maximum class size with the log of the actual class size. Here I have observed a trend that appears to be linear. The number of DAGs to which I calculated the equivalence class size is not very large, and a larger, more diverse set may shed some light on the matter, but I think that I am simply not looking at the correct features.

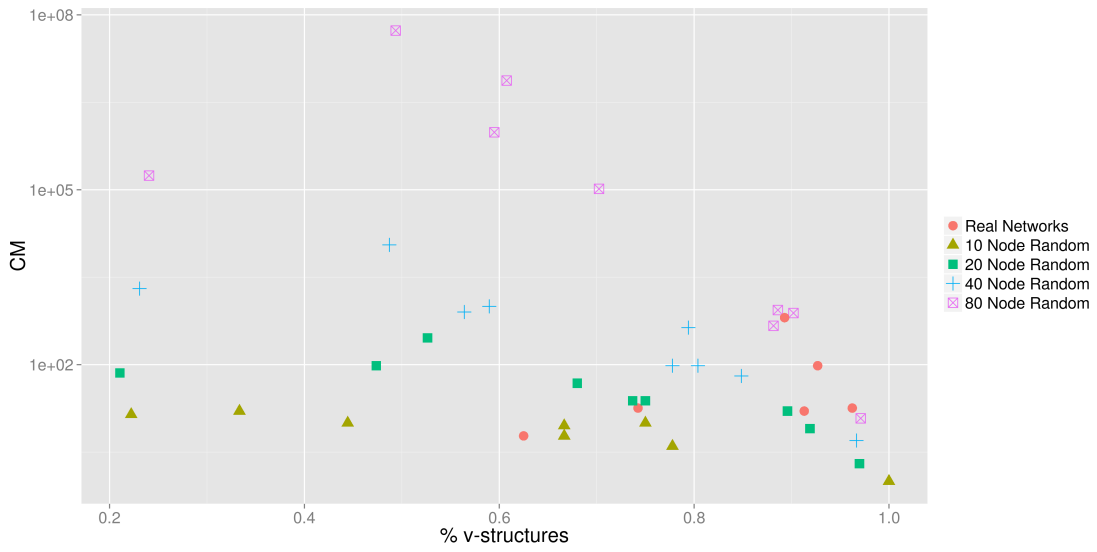


Figure 3.9: Actual size of the equivalence class of each of the DAGs plotted against the percentage of v-structures in the DAG

3.4 DISCUSSION

On theoretical grounds, pattern measures should be preferred over DAG measures. However, my experiments show that the measurements performed either on DAGs or patterns are similar, although cases exist in which distance measures applied to DAGs or patterns differ significantly.

This may be due to a specific interpretation of some of the measures, but a possible explanation is that, in patterns more errors are made due to the fact that more different edges are possible. From the theoretical point of view, this is counterintuitive, since a pattern represents a DAG equivalence class that can contain many DAG structures, which should nullify many small errors that should negatively impact measures performed on DAGs. A possible conclusion is that DAG measures are underestimating the true distance between the learned and the original DAGs.

I observed, surprisingly, that for several cases having more data available for the learning algorithm resulted in DAG structures that were more distant from the original DAG. This

affected both DAG and pattern structures. I assume that the cause is that specific structure learning algorithm I used for the experiments.

One could expect that a large proportion of v-structures in a DAG should have a positive effect on the performance of the structure learning algorithm. However I found the opposite: The average Hamming distance increased with the percentage of v-structures in the DAG. One possible explanation of this observation is that when an algorithm makes one mistake, either wrongfully cutting an edge or later wrongfully orienting one, it may have a large influence on the whole structure, causing many other edges to be directed in the wrong way [Dash and Druzdzel, 1999]. When there are many v-structures to detect, one mistake may cause errors in other v-structures. Finally, I calculated the sizes of equivalence classes of all the DAGs used in my experiments and two more large networks from the literature. These sizes were surprisingly small, which I find consistent with the above hypothesis.

When there are few, the problem may not have a large effect on the end result. The Greedy Thick Thinning and Bayesian search algorithms do not seem to have this problem, being based on the search and score principle, and deal with edge orientation in a way that does not rely on applying edge orientation rules like PC does.

My main finding is that there are no significant differences between comparing DAGs and comparing equivalence classes of DAGs. v-structures have a significant but unexpected influence on the end result: their abundance in a DAG does not seem beneficial for learning algorithms.

One robust conclusion of my experiments is that relying on only one measure of distance is risky, as the measures do not always yield the same ordering of graphs. In addition, aggregate measures, like the Hamming distance hide potentially interesting differences between discovery algorithms and may not give a complete picture. I recommend, wherever possible, to report several measures of distance, both between DAG and patterns, and where applicable other types of quality measures if the problem domain allows this (i.e. classification).

4.0 PARALLELIZING BAYESIAN NETWORK STRUCTURE LEARNING ALGORITHMS

In this chapter I review related work on parallelizing algorithms applied to Bayesian network and discuss opportunities for parallelization.

4.1 INTRODUCTION

With the advancement of technology, the availability of parallel or distributed computation capacity has increased significantly. Through services such as Amazon Cloud, parallel computation has become available to just about everyone. It has become much more feasible to address problems of a much larger magnitude than before.

Initially, parallel computation was mostly based around the MPI (message passing interface) programming model [Snir et al., 1995, Gropp et al., 1999], but with the development of the MapReduce model, attention has shifted more to the latter due to the ability of the MapReduce programming model to abstract away most of the technical details of managing the underlying computational hardware.

Before proposing my own set of parallelized algorithms for Bayesian network structure learning, I review previous work that used either the MPI or the MapReduce model as basis for the parallelization of BN structure learning algorithms. Next, I summarize and discuss possible strategies for parallelization for the main classes of structure learning algorithms. These will serve as a starting point for the work proposed in Chapter 5.

4.2 BACKGROUND

4.2.1 Message Passing Interface

The Message Passing Interface (MPI) [Snir et al., 1995, Gropp et al., 1999] is a standardized approach to parallel computation. It allows users to define topologies for computer clusters and dictate how messages are passed among the computers. Library implementations of the MPI protocol are freely available and abstract away some of the intricacies of the communication process for computer clusters. A good amount of work discusses parallel algorithms for Bayesian network structure learning using the MPI paradigm. Most of the papers are parallelizing variants of the search-and-score approach, some of them learn a Bayesian network as a part of algorithms used for the Evolutionary computation (EC) paradigm. I found only one paper that discusses a parallel constraint-based approach.

Three papers discuss a straightforward parallelization of search-and-score; Xiang and Chu [1999] present an search-and-score approach with multi-link lookahead to learn decomposable Markov networks (DMNs), but show how this approach applies to Bayesian network structure learning. Lam and Segre [2002] discuss a distributed search for Bayesian network structures using the minimal description length (MDL) as score function. The search space is divided over the computers in the cluster and a asynchronous mechanism known as *nagging* is used to prune the search space when possible. Liu et al. [2005] describe a structure learning algorithm for an extension to Bayesian networks known as module networks. Module networks assume that variables can be partitioned in sets that variables within these sets have similar (conditional) dependencies and show similar probabilistic behavior. This effectively allows sharing of parents and parameters. Using a decomposable Bayesian score function, the evaluation of neighboring network structures is distributed over multiple processors.

Both Nikolova et al. [2009] and Tamada et al. [2011] present parallel algorithms for exact Bayesian search, where a dynamic programming algorithm is used to find the global optimal network structure. While Nikolova et al. [2009] start from the beginning, build the parallelized DP algorithm from the ground up, and map their calculations to a hypercube, Tamada et al. [2011] present a more direct parallelization of the original DP algorithm. Due

to the enormous size of the structure search space, both approaches can still only handle a limited number of variables (about 20-30). Both, however, do improve upon the efficiency of the search process.

Yu et al. [2007] present a parallelized version of the Structural EM algorithm [Friedman and Koller, 2000]. This makes this the only approach capable of learning a complete BN model with missing data. The focus of parallelization is on the EM algorithm that is being used to calculate the expected statistics using the initial network structure to fill up the gaps of the missing data. With a now complete dataset, the score calculation is again decomposable and the usual search strategies can be applied to find new candidate network structures.

Sahin and Devasia [2007] discuss a Bayesian search algorithm built around particle swarm optimization that makes use of the K2 metric. Each particle represents a Bayesian network structure and is encoded with a string of 0's and 1's, where a 1 encodes an arc between two nodes. The strings effectively serve as coordinates in a space through which the particles are moving. Every update to the particles moves them to another location in the search space making use of random changes to the particle coordinates. The fitness calculation for each of the particles (using the K2 metric) is done on a separate processor, after the calculations are completed they are sent back to a master processor which advances all the particles to their next point in the search space.

All the work discussed up to this point has an end goal to produce a Bayesian network structure and all use a score-and-search algorithm. I have come across work in the evolutionary computation field where Bayesian networks are being used as a tool for representing a population distribution used for sampling a new population for the next iteration of the EC algorithm. Očenášek and Schwarz [2000] describes a parallel Bayesian optimization algorithm for the field of estimation of distribution algorithms (EDA). These algorithms are similar to genetic algorithms (GA), but instead of having operators such as mutation and crossover a distribution over the solution population is derived and this distribution is used as sampling tool to generate the next population of possible solutions. To represent the population distribution the authors apply a Bayesian search algorithm to learn a BN structure that represents the distribution. For each iteration the authors generate a node order and

distribute the search for optimal local family structures over a number of computers. Munetomo et al. [2005] present a similar approach but do not assume a node ordering and thus have had to add mechanisms to prevent the occurrence of cycles when merging the optimal node families into the final Bayesian network structures. The authors accomplish this by making use of a communication protocol that broadcasts rollback requests to computers in the cluster when cycles are detected. The approach described by Mendiburu et al. [2006] is quite similar to the work by Munetomo et al. [2005], but the Bayesian information criterium is used as a score and the local scoring tasks are not just distributed over multiple computers but over multiple threads in each computer as well.

The final work to be discussed is a parallel implementation of the most computationally critical elements of MMHC (Max-Min Hill-Climbing), a constraint-based / search-and-score hybrid algorithm by Tsamardinos et al. [2006]. The algorithm, presented by Nikolova and Aluru [2011] parallelizes the main bottlenecks of the MMHC algorithm: 1) the discovery of candidate parent and children (CPC) sets for each of the variables, and 2) the calculation of the Markov boundaries for each of the variables. The CPCs for each variable is calculated on separate processors (when available), storing intermediate calculations for reuse in the second phase, where the CPCs are adjusted so they represent the proper Markov boundaries of each variable. The end result of the algorithm is the undirected skeleton of the BN, which would then typically be oriented using a heuristic search.

4.2.2 MapReduce

There seems to be only a limited amount of literature available on parallel Bayesian network structure learning algorithms that use the MapReduce programming model. I found only two papers that discuss a MapReduce version of a BN structure learning algorithm. Chen et al. [2011] propose a parallelized version of the Three-Phase Dependency Analysis (TPDA) algorithm by Cheng et al. [1997]. Fang et al. [2013] propose a MapReduce version of a Bayesian search algorithm that uses the K2 metric for scoring candidate structures.

Chen et al. [2011] describe a BN learning algorithm used for sentiment analysis of unstructured text. The authors have developed a MapReduce version of the TPDA algorithm.

They have focused their efforts on parallelizing the (conditional) mutual information calculations that are central to the algorithm:

$$MI\left(A,B\right)=\sum_{a,b}P\left(a,b\right)log\frac{P\left(a,b\right)}{P\left(a\right)P\left(b\right)} \tag{4.1}$$

$$MI\left(A,B\left|C\right.\right)=\sum_{a,b,c}P\left(a,b\left|c\right.\right)log\frac{P\left(a,b\left|c\right.\right)}{P\left(a\left|c\right.\right)P\left(b\left|c\right.\right)}. \tag{4.2}$$

Their approach consist of a central controller which employs MapReduce jobs to acquire the necessary mutual information scores to be able to run TPDA. TPDA has three phases: 1) drafting, 2) thickening, and 3) thinning. Each of these phases requires MapReduce jobs to provide the mutual information scores necessary for adding, or removing arcs/edges from the graph. Essentially, the mutual information calculation boils down to counting (co-) occurrences of variables in the dataset. As is typical for MapReduce implementations, the records of the dataset are distributed over the Mappers and each Mapper processes one data record at a time.

From each record, the Mappers construct multiple key-value pairs. Each key represents a set of variable assignments, and the value is simply the value '1'. One example of such a constructed pair is $\left(\left(A=a_1,B=b_2\right),1\right)$. All the constructed pairs are collected and then sorted and grouped by their keys. The groups are then assigned to separate Reducers which sum the '1's for each key to acquire the counts for each of the variable assignment sets.

After a MapReduce job finishes, the central controller can collect all the counts and perform the mutual information calculations. In their empirical evaluation, the authors show that the dataset needs to be sufficiently large before the MapReduce approach outperforms a non-parallelized version of the TPDA algorithm. This is due to the extra overhead required by the MapReduce framework.

The MapReduce Bayesian search algorithm proposed by Fang et al. [2013] assumes that for each node an optimal family of parents can be found in parallel and that the final, optimal network can be created by merging the local optimal structures. This assumption depends on a node ordering to be provided to the algorithm at the start. The authors do not discuss how such a node ordering is obtained and assume it is given. Their MapReduce parallelization of the sufficient statistics necessary for the K2 calculation seems to be based

on the work by Chen et al. [2011]. The experimental setup as described by the authors appears to be quite limited. The authors have only considered data from the Asia network, which has only 8 nodes. This dataset was possibly not complex enough to demonstrate any issues which may occur when the number of variables in a dataset increases.

4.3 PARALLELIZATION OPPORTUNITIES: SEARCH BASED ALGORITHMS

Considering the related work that I reviewed in the previous section, the main focal points for parallelization of search and score algorithms are: 1) the calculation of the score function, and 2) efficiently navigating the structure space Additionally, both of the points are influenced by the dimensions of the dataset. Next, I am summarizing and discussing possible strategies for algorithm parallelization, looking at the influence of the size of the dataset and at the main components of search and score algorithms.

4.3.1 Dataset Size

The size of the dataset has a large influence on how we approach the problem of creating a parallel algorithm. If the dataset completely fits in the local memory of the Mappers, we can employ a much larger array of parallelization strategies to cope with the size of the structure space. If the dataset is too large, we may have to explicitly parallelize the calculation of the sufficient statistics and the structure score calculation itself.

Given that the datasets fits in the memory of the Mappers, one trivial approach to Bayesian search parallelization is to run separate instance of the algorithm in each of the Mappers, providing a different random starting point for each of the Mappers. After completing the search, each Mapper outputs its best structure and the corresponding structure score. The Reducer then finds and returns the structure with the maximum score.

If the dataset does not fit in the Mapper memory, these types of approaches are not feasible. To be able to calculate the structure score, an approach similar to Chen et al.

[2011] will be necessary to first calculate counts from the datasets using MapReduce jobs. This way a central controller could construct the Bayesian network in the same manner as a typical sequential algorithm, but then offload the expensive calculation on the dataset to the MapReduce cluster.

A different approach to solving the problem of large data is to select representative samples of the original dataset, which could potentially fit into Mapper memory again. Another potential approach is to partition the columns of the dataset and apply structure learning algorithms to the partitions in the Mappers and merge the substructures in the Reducers.

4.3.2 Score Calculation

Calculating the score of a candidate structure is the most computational-intensive part of any search-and-score algorithm. As was discussed in Section 4.3.1, the size of the datasets that are to serve as input, will guide the design process of the parallel versions of the algorithms.

The decomposition property of most popular score functions is a useful starting point for parallelization, with the added assumption of a fixed node ordering, we can perform the search for the optimal network structure completely in parallel, given that each Mapper has access to the complete dataset, and merge the substructures into the final Bayesian network.

If the dataset is too large, approaches like the one described by Chen et al. [2011] are a natural choice. It is theoretically possible to calculate all the necessary counts in one MapReduce job, but it is quite likely that we cannot store all the intermediate key-value pairs (KVP). Assuming, as was the case for [Chen et al., 2011], we are calculating (conditional) mutual information, then we can calculate the (conditional) mutual information in the following number of ways:

$$\#MI = \binom{C}{2} \cdot 2^{C-2}\,, \tag{4.3}$$

where C is the number of columns in the dataset. This assumes that there is no limit on the size of the conditioning set. Luckily we do not require as many KVPs for the required counts, as many of the counts can be reused for the different mutual information queries,

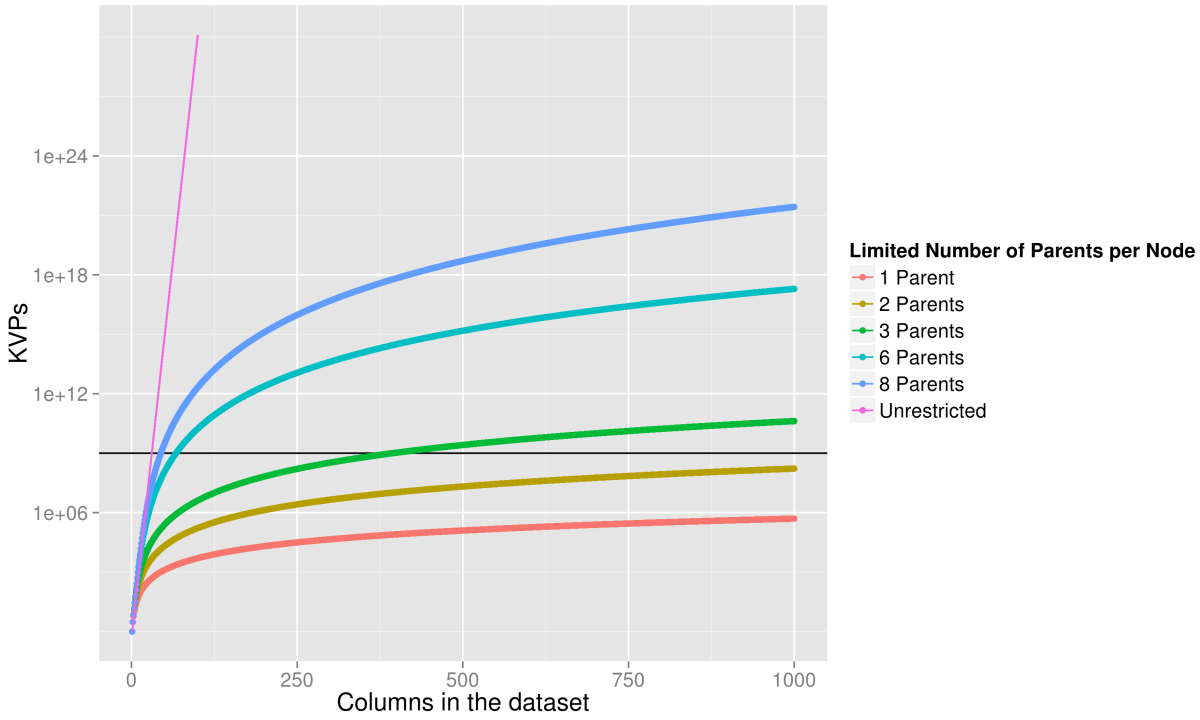


Figure 4.1: Number of KVPs emitted by Mappers for each record

but the total number will still be unacceptably large when the number of columns in our dataset increases:

$$\#KVP = 2^C - 1\ . \tag{4.4}$$

For the Bayesian score functions the number of necessary KVPs stays the same. In the extreme case we would consider a node with all other nodes as potential parents. Typically, we restrict the number of parents and this will restrict the number of KVPs. Assuming we are examining for every node the potential parent set, setting the maximum allowed number of parents to P, we find the total number of KVPs to be outputted for each record in the dataset:

$$KVP = \sum_{i=1}^{C} \sum_{p=0}^{\min(C-i,P)} = \binom{C-i}{p}\ . \tag{4.5}$$

Figure 4.1 shows a plot of the function for a few values of P and dataset sizes up to 1,000 columns. The figure shows that limiting the number of parents is a necessity to keep this approach feasible for anything but toy examples. The unlimited case, examining every

possible parent set for each of the nodes, would require an exponential number of KVPs to be generated for each of the data records. Limiting the number of parents allows for many orders of magnitude less KVPs to be generated, but even then the number of KVPs quickly increases with the number of columns in the dataset. To keep the KVPs from increasing too rapidly, the maximum number of parents needs to be restricted to roughly 3-6 parents per node. The horizontal line in the plot shows where each Mapper would be generating one billion KVPs for each record. It's likely that this will put a considerable amount of load on the MapReduce cluster, but the end result could still be manageable. The number of columns in the dataset will have stay quite small if we want to apply this approach, 50-375 columns might be acceptable at most. The magnitude of the result, all the collected counts, of the MapReduce job will require, for any further score calculation, that a MapReduce implementation is explored for the next part of the score calculation. This issue has not yet been addressed in the literature and thus allows for a possible avenue of research. Other ideas to be considered on the basis of these findings are, as already discussed in Section 4.3.1, examining separate column partitions or subsamples of the data.

An alternative solution to is to "serialize" the problem. Instead of attempting massive MapReduce jobs that calculates all counts for all nodes at the same time, a MapReduce job could consist of just one specific calculation. This could mean calculating the counts for one (conditional) mutual information measure [Chen et al., 2011], or one Bayesian score for a node with a specific set of parents [Fang et al., 2013]. The downside of this approach is that MapReduce jobs tend to have a significant amount of overhead, typically taking at least 30 seconds to finish. If the algorithm requires many MapReduce jobs, the total execution time of the algorithm may potentially become too large for practical applications.

4.3.3 Structure Search

Assuming that we have dealt with the issue of score calculation, we can examine possible parallelization strategies for the structure search. For convenience we will assume that the data fits in memory or that we have easy access to the necessary counts and we can compute the structure scores locally in each Mapper. The main examples from the literature are 1)

assuming a node ordering and find all the optimal node family structures separately, and 2) systematically divide up the structure search space over the processors. Let us consider an alternative.

An iterative algorithm can be constructed by having a central controller work on the final network structure and creating MapReduce jobs where the input for the Mappers are sequences of modifications that are to be applied to the current network structure. Each Mapper can apply the provided modification sequence and re-score the structure after every change. The Mappers can then return the structure with the highest score that it encountered while modifying the originally supplied structure. Reducers can be used to pick the best structure overall the result of the MapReduce job is used by the central controller to update the current structure, after which a new MapReduce job can be created to repeat the process. To be effective the generated sequences should be long enough so that there are enough of them to consider so the overhead of the MapReduce jobs will not dominate the computation time. The algorithm may also apply random starting points to be able to explore a larger area of the search space.

Building on this principle, an algorithm can be defined that performs many searches, each from a randomly generated starting point. Each Mapper could contain an instance of a Bayesian search algorithm, performing a search from the provided starting point, making use of possibly different search heuristics such as tabu search or more random restarts. Reducers would be used to find the maximum scoring structures and possibly could be used to find similarity between high scoring networks, which then could be provided as structural constraints if multiple MapReduce jobs are run instead of just one search iteration.

4.4 PARALLELIZATION OPPORTUNITIES: CONSTRAINT-BASED ALGORITHMS

In the literature I found just one paper [Nikolova and Aluru, 2011] that discussed parallelizing an algorithm related to the constraint-based search approach. Here I will consider possible approaches to parallelizing the PC algorithm [Spirtes et al., 1993]. The two main phases of

the PC algorithm are 1) determining the skeleton of the Bayesian network, and 2) orienting the edges of the skeleton. Let us first look more closely at the independence tests.

4.4.1 Independence Tests

Since the first step involves determining the (conditional) independence between pairs of nodes, potentially involving a conditioning set, again the size of the dataset will play large role in how we parallelize the algorithm. If the dataset does not fit in memory we will again have to employ the same strategy as discussed before, but due to the ordering of the (conditional) independence tests, we can this time adapt an iterative approach where we start out with a MapReduce job that calculates the pairwise independence among variables and the removes edges between variables deemed independent before continuing with MapReduce jobs that calculate conditional independence among the variables. Every iteration we would prune the edges that are independent and then we increase the size of the conditioning, where the number of tests will be reduced due to the decrease in unsuccessfully tested edges. Combined with limiting the maximum number of variables in the conditioning set, the number of KVPs emitted for each data record should be greatly reduced, making the procedure more feasible for larger datasets and a potentially interesting preprocessing step for parallel Bayesian search algorithms.

If the dataset does fit in Mapper memory, we can perform each independence test completely within a Mapper and apply a different parallelization strategy. The independence tests, can be divided over the Mappers. Potentially tests for the same node pair, but with different conditioning sets can be distributed over Mappers as well, but this would require more complicated logic in the Reducers.

4.4.2 Edge Orientation

The edge orientation phase of the PC algorithm is inherently iterative, and this complicates its parallelization when using the MapReduce framework. The framework is less suited for iterative algorithms. Orienting the edges is done in two steps. First we search for v-structures, and after this has been completed we proceed to orient the remaining edges by

matching them with a set of rules. The nature of the v-structure discovery step allows for parallelization. We examine all triplets of nodes looking for ones that satisfy all of the following conditions for each triplet (A, B, C):

- There must be an edge between A and B
- There must be an edge between B and C
- There must not be an edge between A and C
- B must not be present in the separator set (calculated during independence tests) of A and C

We can examine all possible triples in parallel, but it would probably suffice to examine every node in parallel instead and have each mapper consider every pair of adjacent nodes. Compared to the calculation of the independence tests, the amount of calculation necessary here most likely does not justify applying the MapReduce framework. The ratio of overhead to actual computation will stay unfavorable unless the number of nodes increases significantly.

The final step of the PC algorithm is to orient the remaining edges. The PC algorithm iteratively applies the following two rules to the graph.

1. Orient $A \to B - C$ as $A \to B \to C$
2. Orient $A - C$ as $A \to C$ if there is a path $A \to \ldots \to C$

Depending on preference any remaining undirected edges are either oriented randomly, but so that the result is a consistent DAG, or are left unoriented and the algorithm returns a pattern. In the SMILE implementation, the two rules are investigated sequentially. First Rule 1 is applied to any valid match then Rule 2 is applied to any match. The algorithm will keep applying the two rules until neither is able to make any new changes to the graph.

Similar to the v-structure search, we can potentially parallelize the rule matching to examine candidates in parallel, but this will most likely not increase algorithm performance due to MapReduce overhead as well. A more interesting idea to parallelization is to allow for multiple rule application strategies. In SMILE, the rules are applied in a strict order. It might be worthwhile to examine the consequences of applying the rules in a different order or when the rules are interleaved, switching between rules after every step. Applying these different rule matching strategies will result in a number of candidate structures. As part of

the reduce job, we can then investigate the similarities and differences and perhaps are able to return a more robust structure. This structure might be less dependent on the artifact of the specific rule ordering as implemented in libraries such as SMILE.

4.5 OTHER PARALLELIZATION LITERATURE ON BAYESIAN NETWORKS

This dissertation has a focus on learning Bayesian network structures. Hence, up to this point, this chapter has only reviewed parallel Bayesian network structure learning approaches from the literature and parallelization opportunities for the existing structure learning doctrines. To round out this chapter, I will discuss parallelized approaches to Bayesian network inference and parameter learning. Section 4.5.1 will discuss inference approaches and Section 4.5.2 will discuss parallelized approaches to parameter learning for Bayesian networks.

4.5.1 Inference

Pennock [1998] described a parallel inference algorithm called McPHISHFOOD, which was designed for the theoretical CREW PRAM, concurrent-read, exclusive-write parallel random-access machine model. The algorithm constructs a directed version of a junction tree and applies often-used techniques for algorithm parallelization such as *pointer jumping*, *parallel tree contraction*, and the *Euler tour technique*, to perform the junction tree inference calculations in logarithmic time.

Namasivayam and Prasanna [2006] have implemented a topology independent parallel version of the junction tree algorithm. The algorithm has been parallelized on two levels: 1) a top level, where pointer jumping was applied for moving between clique nodes, and 2) a node level parallelization used to exploit independences in the node calculations. Their approach is limited to only one evidence variable, which should appear in the root of the junction tree. The algorithm was implemented using the MPI library.

Xia and Prasanna [2008] present an algorithm for exact inference that decomposes a junction tree into a set chains. In contrast to the [Namasivayam and Prasanna, 2006] algorithm, this algorithm is able to process evidence in multiple variables, regardless of their location in the decomposed junction tree. To incorporate evidence, the cliques are updated partially first in their chains, and are then merged to construct the fully updated cliques for the complete tree. Evidence is propagated between the chains and the nodes using the pointer jumping technique. MPI was used to implemented and run the algorithm on a cluster.

Ma et al. [2012] describe an approach to parallel inference of Bayesian networks using the MapReduce framework. Their algorithm was not on a Hadoop cluster with separate computers, but on a multi-core computer with an additional software layer that facilitates the Hadoop framework on a single computer with many CPU cores. Just as the other papers discussed in this section, the authors have proposed a parallel version of the junction tree algorithm. Specifically they propose two strategies for evidence propagation. The first approach is a depth first search approach, where MapReduce jobs consist of clique calculations in Mappers and Evidence adsorption in Reducers. MapReduce calculation are in the strategy limited to one node and it's children. The other strategy proposed by the authors intends to exploit a larger level of parallelism by performing calculations for whole tree layers, i.e., by collecting evidence from all leaf nodes and propagating it to their parents, all in one MapReduce job. The authors found that the level-based strategy was the better performing one of the two they proposed.

4.5.2 Parameter Learning

The most common algorithm used for learning the parameters of a Bayesian network is the EM algorithm [Dempster et al., 1977, Lauritzen, 1995]. Although the EM algorithm is specifically meant for learning parameters when the data are incomplete, it is commonly used in the complete data case as well, where it reduces to maximum likelihood estimation.

Yu et al. [2007] actually describe a parallel structure learning algorithm, but it is based on the structural EM algorithm by Friedman and Koller [2000]. To learn the model parameters, they apply their parallel version of the EM algorithm, PL-SEM. The authors apply data-

parallelization to calculate the sufficient statistics. The dataset is distributed equally over the computers for this step. The parameters for the BN are estimated in the M-step of the algorithm, the CPTs of the network are distributed over the computers to perform this step in parallel. The authors have used the MPI library to implement their algorithm.

Mensink et al. [2007] present a distributed version of the EM algorithm for the problem of multi-camera tracking of persons. They authors apply mixture of Gaussian to represent the persons under investigation. Their graphical model used for tracking is a special case of a Bayesian network with continuous variables. Their algorithm, Multi-Observations Newscast EM (MON-EM), performs local EM computations at each camera and the information at each camera is communicated with the others. Specifically, the E-step is identical to the regular EM algorithm, but the M-step involves what the authors call gossip-based cycles where cameras randomly contact another camera and exchange information. After a number of cycles the local mixture of Gaussians model at each camera converges on the correct, global parameters.

Basak et al. [2012] present a MapReduce version of the EM algorithm. One MapReduce job represents one iteration of the EM algorithm. The authors provide optimized versions for the complete and incomplete data case. The complete case reduces to the basic "generate counts and add them up" approach. The Mapper creates counts for variable assignments in the counts and the Reducer adds these up. A central controller then processes the counts and updates the parameters of the Bayesian network. The incomplete case involves performing inference on each of the samples in the Mapper to estimate the sufficient statistics of the missing data. We now deal with "soft" counts, which are distributions over all possible states the variable can be assigned. These distributions are aggregated in the Reducer and the new parameters for each CPT entry are determined here and outputted to a central file to update the BN. The process continues until the parameters converge.

5.0 UTILIZING MAPREDUCE TO REALIZE LARGE SCALE USE OF THE PC ALGORITHM

In this chapter, I propose two MapReduce algorithms based on the PC algorithm. I evaluate these algorithms together with two algorithms from the literature and a single-processor implementation of the PC algorithm.

5.1 INTRODUCTION

In Chapter 4, I discussed related work on parallelizing Bayesian network structure learning algorithms based on either the MPI or the MapReduce framework. I also summarized and discussed possible approaches to parallelizing structure learning algorithms using MapReduce. In this chapter, I describe two MapReduce variants of the PC algorithm and provide a more detailed explanation on two MapReduce algorithms for learning Bayesian network structures from the literature. I report the result of an empirical evaluation of the performance of the four algorithms and SMILE's PC algorithm on a set of gold standard network recovery tasks and a set of datasets for a classification task. I conclude this chapter with a discussion of the performance of each of the algorithms and a general discussion of the feasibility of learning Bayesian network structures with the MapReduce framework.

I found, after running evaluations, that it is challenging for Hadoop-based algorithms to compete with a well-optimized single-processor algorithm such as the PC algorithm, available in the SMILE library. The reference algorithms from the literature [Chen et al., 2011, Fang et al., 2013] claim a focus on datasets with many records, but the true difficulty of learning Bayesian network models lies in the number of variables that datasets have. The algorithms

I propose are based on the PC algorithm. A MapReduce version of the PC algorithm has the potential to eliminate a large amount of the computational complexity early in the process, if the data are suitable. Any MapReduce job has a significant amount of overhead and each one takes at least seconds to complete. Algorithms that require many of them will be significantly slower than those that require less or none at all (e.g. SMILE).

I found, that only when the number of variables in a dataset becomes truly large, starting around 800 variables, my second MapReduce algorithm starts to outperform the single processor SMILE code, when the sheer number of Mapper jobs executed in parallel starts to make a difference.

5.2 REFERENCE ALGORITHMS

To provide a frame of reference for the evaluation of my two MapReduce-based PC algorithm implementations, I implemented the algorithms described by Chen et al. [2011] and Fang et al. [2013]. Both are MapReduce-based algorithms for learning Bayesian network structures, albeit based on different non-MR algorithms. The algorithm by Chen et al. [2011] is based on the Three-Phase Dependency Analysis (TPDA) algorithm by Cheng et al. [1997] and the work by Fang et al. [2013] was based on the classic Bayesian search algorithm by Cooper and Herskovits [1992]. When implementing the algorithms, I followed the respective papers as closely as possible. To implement the work by Fang et al. [2013], I made a few choices to fill in the gaps of the description in their paper. In the following two sections, I will reproduce the algorithms, as they were implemented by me, and summarize how they work.

Finally, in Section 5.2.3 I describe a few details of the implementation of SMILE's PC algorithm, which was used as a third reference algorithm to compare the performance of the MapReduce-based algorithms to a highly optimized Bayesian network structure learning algorithm running on a single computer.

5.2.1 Chen's Algorithm

The algorithm proposed by Chen et al. [2011] follows closely the original structure of the TPDA algorithm it was based on. It consists of a main controller process that executes all the steps of the algorithm. It creates MapReduce jobs to acquire the counts that are necessary for the calculation of the (conditional) mutual information measure used for its edge removal decision making process. The TPDA algorithm consists of four phases:

1. Drafting
2. Thickening
3. Thinning
4. Edge Orientation

In the first phase, the algorithm calculates all the pairwise mutual information measure between all nodes. A preset threshold is used to decide which edges to keep. The first phase results in an initial skeleton. The second phase attempts to add edges using conditional mutual information (CMI) measures between nodes and conditioning variables. The third phase then tries to prune the network again using CMI. The final phase is identical to the edge orientation phase of the PC Algorithm [Spirtes et al., 1993].

The MapReduce part of the algorithm is used purely for calculating counts. The algorithm employs two different types of jobs. The first calculates counts needed for pairwise mutual information calculations for all possible node pairs. The second, calculates counts necessary for a specific conditional mutual information measure for nodes X and Y given conditioning set $\mathbf{S}$. They differ in their Mapper code, but both use the same Reducer code, which simply adds up all counts for a key. Algorithms 3 and 4 describe the two Mapper implementations and Algorithm 5 describes the Reducer code, which is identical to the Reducer code used for adding up the counts in my first MapReduce PC algorithm.

The main controller code calls the Hadoop jobs as subroutines. I present the details in Algorithm 6. The algorithm has been implemented as a standalone class, it can be provided to Hadoop boilerplate code and run on any Hadoop cluster.

Algorithm 3 Mutual Information Counts: Mapper

Split input record in separate column values
for each column X in the record **do**
  Construct $v_X = val(X)$ assignment
  Output this assignment with count set to 1
  for each column Y in the record **do**
    Construct output assignment $v_X = val(X) + v_Y = val(Y)$
    Output this assignment with count set to 1
  end for
end for

Algorithm 4 Conditional Mutual Information Counts: Mapper

Split input record in separate column values
For pair (X, Y) and conditioning set $\mathbf{S}$:
Construct $v_X = val(Y)$ assignment A
Construct $v_Y = val(Y)$ assignment B
Construct $v_{S_1} = val(S_1) + v_{S_2} = val(S_2) + \ldots v_{S_n} = val(S_n)+$ assignment C
Output $(A + C, 1)$
Output $(B + C, 1)$
Output $(A + B + C, 1)$
Output $(C, 1)$

Algorithm 5 (Conditional) Mutual Information Counts: Reducer

$total = 0$
for every input $(key, count)$ **do**
  Add *count* to *total*
end for
Return $(key, total)$

Algorithm 6 Main controller

```
[Drafting Phase]
Calculate all pairwise MI between nodes (MapReduce MI),
keep only edges with MI > threshold in list L
for each edge (X, Y) in L do
  if there are no paths from X to Y then
    Add the edge (X, Y) to the graph
    Remove (X, Y) from L
  end if
end for
[Thickening Phase]
for each edge (X, Y) in L do
  if the edge (X, Y) is needed (MapReduce, CMI) then
    Add the edge (X, Y) to the graph
  end if
end for
[Thinning Phase]
for each edge (X, Y) in the graph do
  if there are paths from X to Y then
    if the edge (X, Y) is not needed (MapReduce, CMI) then
      Remove the edge (X, Y) from the graph
    end if
  end if
end for
Orient Edges using the rules used by the PC algorithm
```

5.2.2 Fang's Algorithm

Comparing the paper by Fang et al. [2013] to the paper by Chen et al. [2011], the former was harder to implement than the latter. The MapReduce code was easy to understand, but

certain design decisions were left out in the description of the algorithm. For instance, the authors mention only indirectly, that the counts calculated in one MapReduce job need to be available to all Mappers in a subsequent MapReduce job. Making the counts available to all Mappers requires them to be distributed to all computers in the cluster that are running Mapper jobs, which could potentially have a significant impact in overall performance of the algorithm. Furthermore, the authors described the main Bayesian search algorithm in less detail and seem to have forgotten to discuss a crucial point: Bayesian search (as described by Cooper and Herskovits [1992]) requires a predefined node ordering. The authors did not discuss how to choose an ordering, either when proposing their algorithm or when discussing their empirical evaluation. The assumption of a predefined node order is crucial for their algorithm, however, for it allows them to optimize the parent sets for each node independently. For the purpose of using the algorithm as a reference algorithm in the evaluation of my PC algorithm implementations, I chose to provide it with a correct node ordering when performing a gold standard recovery task.

The algorithm makes use of three different types of MapReduce jobs. The first two are for calculating counts, where one (Algorithm 7) is specialized for the case where we need counts for a node with an empty parent set, and the second (Algorithm 8) is used when the parent set is non-empty. Both use the same Reducer code (Algorithm 9). The last MapReduce job (Algorithms 10 & 11) is used for selecting the best new parent candidate to add to the parent set of a node. The main controller for the algorithm is listed in Algorithm 12.

Algorithm 7 Counts for Parent-Less Node: Mapper

1: For node I:
2: Construct assignment $v_I = val(I)\ A$
3: Output $(A, 1)$

Like Chen's algorithm, this algorithm has been implemented as a standalone class, that can be provided to Hadoop boilerplate code and run on any Hadoop cluster. A detailed description of the practical performance of both algorithms is given in Sections 5.6.3 and

Algorithm 8 Counts for node with Parents: Mapper

For node I:
Construct assignment $v_I = val(I)$ A
Construct $v_{S_1} = val(s_1) + v_{S_2} = val(S_2) + \ldots v_{S_n} = val(S_n)+$ assignment B
for all parent candidates Y **do**
    Construct assignment $v_Y = val(Y)$ C
    Output $(C + B + A, 1)$
end for

Algorithm 9 Counts: Reducer

$total = 0$
for every input $(key, count)$ **do**
    add $count$ to $total$
end for
Return $(key, total)$

Algorithm 10 Find Best New Parent: Mapper

Get node I
Get current parent set $\mathbf{S}$
Get counts necessary for score calculation
Read input parent candidate Y
Calculate score for node I given parent set $\mathbf{S} \cup Y$
Output $(Y, score)$

5.6.4 respectively. The main difference between my algorithm implementations and these two algorithms is that these implement smaller parts of the algorithm as MapReduce jobs. Consequently a complete run of either algorithm requires many MapReduce jobs to be run to complete all the work. This design choice effects the scalability in terms of the number of variables in the dataset.

Algorithm 11 Find Best New Parent: Reducer

```
1: max = 0
2: best_candidate = 0
3: for every input (candidate, score) do
4:   if score > max then
5:     max = score
6:     best_candidate = candidate
7:   end if
8: end for
9: Return (best_candidate, max)
```

Algorithm 12 Main controller

```
1: for each node i do
2:   Calculate Score for i with an empty parent set S (MapReduce Job)
3:   OkToProceed = true;
4:   while OkToProceed = true and |S| < max set size do
5:     Find best new parent to add to S (2x MapReduce Jobs)
6:     if new_score > current_max_score then
7:       Current_max_score = new_score;
8:       Add new parent to S;
9:       Add arc (new_parent, i) to the graph
10:    else
11:      OkToProceed = false
12:    end if
13:  end while
14: end for
```

5.2.3 SMILE's PC Algorithm

SMILE's implementation of the PC algorithm was based on the description in the book by Spirtes et al. [1993]. Additionally, several optimizations have been added to the algorithm.

SMILE caches counts obtained from the dataset in a ADTree data structure [Moore and Lee, 1998]. The use of the ADTree data structure speeds up the algorithm significantly, but it requires a large amount of memory. I have added the ability to constrain the memory usage of the ADTree, to ensure optimal performance. If the memory usage is not constrained, it is possible the program will either crash due to running out of memory, or algorithm process may be slowed down to a crawl if the operating system starts swapping out memory to disk. Once the OS starts storing parts of the ADTree to disk, its speed benefit disappears, as the process running the algorithm will spend a considerable amount of time waiting for disk operations to finish. Constraining the memory usage so the ADTree will fit in memory ensures that the algorithm will perform much better, even though, from time to time the ADTree is completely deleted and rebuilt.

To improve the efficiency of the independence testing procedure further, I have implemented a Mutual Information heuristic that reorders the nodes so that edges that are considered the weakest are tested first. The G^2 statistic, used for the significance testing of edges, is proportional to the mutual information measure $MI\,(A, B\,|\mathbf{Z}\,)$. To be exact:

$$MI\,(A, B\,|\mathbf{Z}\,) = \frac{G^2}{2N \log\,(2)}\ . \tag{5.1}$$

Thus, it is trivial to acquire the mutual information for an edge (given conditioning variables if necessary) after we have calculated the G^2 statistic. The mutual information measure is used as a measure of link strength and node strength (by adding up the strength of the links of a node). We sort the nodes so that we evaluate the weakest links first. Potentially this can speed up the process due to weaker links being pruned quicker and not being a part of a later conditional independence test. I have implemented the MI edge pruning ability and the "test the weakest first" strategy (as described in [Bacciu et al., 2013]).

Another optimization implemented for SMILE reduces the size of the conditioning sets used in the independence tests. Based on descriptions in [Spirtes et al., 1993] and [Cheng et al., 1997], I have implemented a procedure that reduces the conditioning set for a test of independence for nodes X and Y to only those nodes that are on paths from X to Y, eliminating effectively nodes that are on "dead ends." This heuristic has the ability to considerably reduce the number of computations needed for conditional independence tests.

I have observed, that in some cases nodes stand out with a very large number of edges, but that these nodes are the center in what resembles a star topology. The majority of its adjacent nodes are only connected to this center node, and with only a few connections to the rest of the network. Eliminating these dead end connections can significantly reduce the number of independence tests that need to be performed. This is due to the number of tests depending on all the possible combinations of adjacent variables as potential conditioning sets.

The final improvement added to the PC algorithm, is a rule that ensures that independence tests are only performed when there is sufficient data available for the contingency tables used for the tests. Spirtes et al. [1993] recommend that on average there are 10 samples per cell, and if this ratio is not met, the test should not be performed and the edge should be kept. Tsamardinos et al. [2006] share this belief, but recommend a ratio of 5 and, although explicitly referring to Spirtes et al. in their work, they recommend the opposite action, i.e., that the edge is removed if there are not sufficient data. This difference of opinion and the potential impact of the two conflicting advices is the topic of Chapter 6. With respect to the three algorithms based on PC that we will consider in this chapter, the PCB algorithm (Section 5.4) and SMILE both follow the recommendation of Tsamardinos et al. [2006]. PCA (Section 5.3), which was implemented before this issue was examined more closely, does not implement this rule.

In the next two sections, I propose two MapReduce algorithms based on the PC algorithm. In section 5.5 I will compare the performance of all four MapReduce based algorithms and SMILE's PC algorithm implementation running on a single computer.

5.3 ALGORITHM: DISTRIBUTED COUNTING (PCA)

My first attempt to a MapReduce version of the PC algorithm was based on the distributed counting principle used both by Chen et al. [2011] and Fang et al. [2013]. In the empirical evaluation section, this algorithm will be referred to as PCA. To acquire the required counts, one MapReduce phase is used to generate the counts from the raw data records. Additionally,

I utilize two additional MapReduce phases that prepare the raw counts for the calculation of the G^2 statistic, the calculation of the G^2 statistic, the p-value, and the selection of edges that are to be removed from the graph. The three phases can be summarized as:

1. Calculate counts
2. Calculate p-values
3. Gather (edge, conditioning set, p-value) triplets with maximum p-value

In the first phase, the necessary counts for the (conditional) independence tests are calculated by applying the same principle as was used by Chen et al. [2011] and Fang et al. [2013]. The Mappers (Algorithm 13) read the data record by record and map the variable assignments to KVPs (Key-Value Pairs). These KVPs are collected and summed by Combiners and Reducers (both apply Algorithm 14) and the counts for all necessary calculations are written to disk at the end of the job.

The next phase (Algorithm 15), takes the counts as inputs and splits the variable assignments into ordered sets of variables, which represents the key and a pair structure that consists of the ordered set of variable outcomes and the count. In the Reduce phase (Algorithm 16), the Reducers collect all counts that belong to the same set of variables. We now have all the information necessary to perform an independence test. Next, we construct the contingency table (only non-zero entries are necessary), calculate the G^2 statistic and the degrees of freedom for the test, and finally calculate the p-value of the test. If a test result is not significant, i.e., the p-value is larger than the preset alpha value, we output the edge and the corresponding p-value.

The final phase (Algorithms 17 & 18) finds the set of edges with the corresponding maximum p-values. This set of edges will be removed from the current graph.

The three-phase Hadoop process is controlled by a central process (Algorithm 19) that runs on the client computer and not on the cluster. I implemented the central controller as a module for the PC algorithm implementation of SMILEWIDE, a Hadoop inspired version of SMILE. Instead of running the independence tests locally, this way a Hadoop cluster can be used without exposing the details of the implementation to the user of the PC algorithm.

Algorithm 13 PCA: Calculate Counts, Mapper

Require: Current graph G

Data record d

Current conditioning set size i

for each column X of d **do**
  for each column Y of d **do**
    if $X \neq Y$ **and** G has an edge between X and Y **then**
      Construct output assignment O:
      "var$X$=val$X$+var$Y$=val$Y$"
      if $i > 0$ **then**
        for $j = 0; j < |\mathbf{PS}(\mathbf{V}(G) \setminus \{X, Y\})|; j++$ **do**
          {We are enumerating over all members of the powerset}
          $\mathbf{C}_j \in PS(\mathbf{V}(G) \setminus \{X, Y\})$
          if $Z \in \mathbf{C}_j \iff Z \in \mathbf{Adj}(X) \vee Z \in \mathbf{Adj}(Y)$ **then**
            Construct conditioning assignment O_j :
            "var1=val1+var2=val2+$\cdots$+var$i$=val$i$"
            Set assignment count to 1
            Output assignment $(O + O_j, 1)$ to Reducer
          end if
        end for
      else if $X < Y$ **then**
        Set assignment count to 1
        Output assignment $(O, 1)$ to Reducer
      end if
    end if
  end for
end for

Algorithm 14 PCA: Calculate Counts, Reducer

Require: all $(key, count)$ pairs with key-value key (Algorithm 13)

1: $total = 0$
2: **for** every input $(key, count)$ **do**
3: $\quad total = total + count$
4: **end for**
5: Write $(key, total)$ to intermediate file

Algorithm 15 PCA: Calculate p-values, Mapper

Require: $(key, count)$ input record from result of previous Reducer (Algorithm 14)

1: Split $(key, count)$ pair into key (variable assignment) and $count$
2: Reformat key:
3: $\quad$ "var1=val1+var2=val2+var3=val3" $\rightarrow$ "var1,var2,var3"
4: $\quad key_{new}$ = "var1,var2,var3"
5: Construct $value$ from $(key, count)$:
6: $\quad$ ("var1=val1+var2=val2+var3=val3",$count$) $\rightarrow$ "val1,val2,val3=$count$"
7: $\quad value$ = "val1,val2,val3=$count$"
8: Output $(key_{new}, value)$ to Reducer

Algorithm 16 PCA: Calculate p-values, Reducer

Require: All $(key, value)$ pairs from Mappers (Algorithm 15)

Construct contingency table CT,
having a dimension for each variable in key,
and for each dimension i have $|Card(var_i)|$ cells
for every $(key, value)$ input **do**
  Decode key ("var1,var2,var3") and $value$ ("val1,val2,val3=$count$")
  into coordinates for CT and store the count:
  $CT\left[val1, val2, \ldots, valn\right] = count$
end for
From CT, calculate G^2 statistic [Spirtes et al., 2000]:
$G^2 = 2\sum_i x_{ijk}\left(\ln\left(x_{ijk}\right) + ln\left(\sum_i\sum_j x_{ijk}\right) - ln\left(\sum_i x_{ijk}\right) - ln\left(\sum_j x_{ijk}\right)\right)$,
where x_{ijk} is a cell in CT, i is a row, j is a column, and k represents a particular instance of the conditioning variables
From CT, calculate degrees of freedom dof [Fienberg, 2007]:
for every conditioning set assignment $\mathbf{C}_i \in \mathbf{C}$ **do**
  $emptyrc\left[\mathbf{C}_i\right] = (row, col)$, where row and col represent the number of empty rows and columns in the contigency table indexed by conditioning assignment $\mathbf{C}_i$
end for
$dof = \left(Card\left(X\right) - 1\right) * \left(Card\left(Y\right) - 1\right) * \left(\left(\prod_i Card\left(C_i\right)\right) - |emptyrc|\right)$
for every conditioning set assignment $\mathbf{C}_i \in emptyrc$ **do**
  $dof_{cor} = \left(Card\left(X\right) - 1 - emptyrc\left[\mathbf{C}_i\right].row\right) * \left(Card\left(Y\right) - 1 - emptyrc\left[\mathbf{C}_i\right].col\right)$
  if $dof_{cor} \leq 0$ **then**
    $dof = dof + dof_{cor}$
  end if
end for
Calculate p-value using χ^2 distribution [Spirtes et al., 2000]: $p = F_{\chi^2}\left(G^2, dof\right)$
if $p > \alpha$ **then**
  Write (key, p) to intermediate file
end if

Algorithm 17 PCA: Gather edge triplets, Mapper

Require: $(key, pvalue)$ pair from results of previous Reducer (Algorithm 16)

Adjust key so p-values for test (X, Y) and (Y, X) are sent to the same reducer:
if $key \rightarrow (X, Y, C_1, C_2, \ldots, C_i) \wedge Y < X$ **then**
  $key = (Y, X, C_1, C_2, \ldots, C_i)$
end if
Output $(key, pvalue)$ to Reducer

Algorithm 18 PCA: Gather edge triplets, Reducer

Require: All $(key, pvalue)$ pairs from Mappers (Algorithm 17)

$max = 0$
$max_{key} = \emptyset$
for All $(key, pvalue)$ **do**
  if $pvalue > max$ **then**
    $max = pvalue$
    $max_{key} = key$
  end if
end for
Decode from max_{key} the conditioning set $\mathbf{C}$:
$(X, Y, C_1, C_2, \ldots, C_i) \rightarrow (C_1, C_2, \ldots, C_i)$
$\mathbf{C} = (C_1, C_2, \ldots, C_i)$
Write $((X, Y), \mathbf{C})$ to output file

Algorithm 19 Main controller for PCA

Require: Completely connected, undirected graph G

1: **for** $i = 0$; $i <$ max conditioning set limit ; $i++$ **do**
2: Run phase 1 (Algorithms 13 and 14) of the Hadoop Process
3: Run phase 2 (Algorithms 15 and 16) of the Hadoop Process
4: Run phase 3 (Algorithms 17 and 18) of the Hadoop Process
5: Download output file created by Algorithm 18
6: **for** each $((X, Y), \mathbf{C})$ in the file **do**
7: remove edges (X, Y) and (Y, X) from G
8: Store $\mathbf{C}$ for later processing in the edge orientation step of the PC algorithm [Spirtes et al., 2000]
9: **end for**
10: **end for**
11: output G to the edge orientation phase of the PC algorithm

After completing all the independence tests, the main controller returns the updated graph and the collection of separator sets to the main PC algorithm code, which will handle the edge orientation process.

In Section 5.6.1, I will give a more detailed report on the performance of the algorithm, but to summarize, I found that this approach could not guarantee successful termination every time. The successful execution of the algorithm depended heavily on the input data. If not enough edges were removed in the first rounds of independence tests, the number of counts, required for the independence tests, would increase exponentially and the total load on the Hadoop cluster would become too much, causing the cluster to run out of disk space and the algorithm to crash. It was necessary for me to investigate alternative approaches to parallelization of the PC algorithm.

5.4 ALGORITHM: DISTRIBUTED INDEPENDENCE TESTING (PCB)

Due to the problems with the PCA MapReduce implementation of the PC algorithm (Mapper message overload), I worked on implementing a second approach, where each Mapper would perform a number of (conditional) independence tests, while having the complete dataset in its memory. In the empirical evaluation, this algorithm will be referred to as PCB. My first implementation of this approach had the Mappers perform independence tests on a random subset of all edges that needed testing. I derived an equation for the probability distribution of missing an edge to determine how many random independence tests the Mappers have to perform in total to keep the error probability (not testing edges) below a preset threshold:

$$P(error) = \left(1 - \frac{K}{E}\right)^{M}, \tag{5.2}$$

where K stands for the number of edges to be tested by each Mapper instance, E the total number of edges present in the graph, and M is the total number of Mapper instances to be run. In this implementation, M and $P\,(error)$ are preset, E is derived from the current state of the graph under investigation and we calculate K using the equation filled in with the other parameters. This means that over time, when we progress to the next iteration and increase the size of the conditioning set, K will become smaller due to the decrease in E through the removal of the edges deemed independent in the previous iteration. It is not uncommon that within a few iterations each Mapper instance will only test one edge for independence.

This, however, does not mean that only one test of independence is performed. Once the conditioning set size is larger than 0, we will perform a test of independence for every eligible combination of conditioning variables. Variables are eligible to be included when they are directly connected to one of the adjacent nodes of the edge under consideration. We do two sets of tests for an edge $X - Y$, one where the conditioning variables must be connected to X, and one where they must be connected to Y. All combinations are tested and PC stores the conditioning set which has the largest p-value (i.e., for that conditioning set it was the hardest to reject independence between X and Y).

After initial testing, I found that the extra amount of work the Mappers had to do to guarantee that the percentage of edges missed in an evaluation would stay low enough, added significant overhead to the process. As a consequence, I abandoned the random approach for a simpler partitioning approach, where the edges were partitioned over a preset number of Mapper jobs. Over successive iterations, removing edges from the graph after each iteration, the number of edges to be processed by each Mapper jobs is adjusted downwards. This ensures that all Mapper jobs will continue to process the same number of edges.

The Mappers (Algorithm 20) output all the edges to be removed, together with the appropriate conditioning sets. All this information is collected by the Reducers (Algorithm 21) and is combined. The end result is collected by the main process (Algorithm 22) that is driving the algorithm. It parses the data and then updates the graph, increases the conditioning set size and starts the next MapReduce job.

Algorithm 20 PCB: Distributed independence tests Mapper

Require: Current graph structure G,
Record of file I with edge testing starting position s,
Number of edges tot test K
Current conditioning set size i

1: **for** $e = s; e < s + K : e++$ **do**
2: **if** edge $e = (X, Y)$ is independent given a conditioning set **C then**
3: add $((X, Y), \mathbf{C})$ to output list **L**
4: **end if**
5: **end for**
6: Output **L** to the Reducers

The general implementation of the PCB algorithm follows the approach of the PCA algorithm. Both are designed to be modules that can be dropped into our PC algorithm implementation. The PCB algorithm proved to be a much more reliable algorithm than the distributed counting variant described in Section 5.3. The algorithm will finish properly given the dataset fits in memory, with sufficient free memory available for the algorithm to

Algorithm 21 PCB: Distributed independence tests Reducer

Require: All (edge, conditioning set) lists $\{\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_n\}$ from the Mappers (Algorithm 20)

$\mathbf{L} = \emptyset$
Merge the lists:
for $i = 1; i < n; i++$ **do**
  for $j = 0; j < |\mathbf{L}_i|; j++$ **do**
    $\mathbf{L}_{ij} \rightarrow ((x, y), \mathbf{C})$
    if $((X, Y), \mathbf{C}) \notin \mathbf{L}$ **then**
      $\mathbf{L} = \mathbf{L} \cup ((X, Y), \mathbf{C})$
    end if
  end for
end for
Write $\mathbf{L}$ to an output file

maintain its data structures. In Section 5.6.2, I will discuss in more depth the advantages and disadvantages of this algorithm.

5.4.1 Correctness of PCB

In this section, I prove that PCB is correct and complete in the large sample limit[1], i.e.,

Theorem 5.4.1. *In the large sample limit, PCB is correct*

Proof. The correctness of PCB follows from 1) by assuming the large sample limit, we also assume that testing independence is now equivalent to consulting a d-separation oracle, which does not make mistakes, and 2) the correctness of the edge orientation phase of the PC algorithm, [Spirtes et al., 1993], which PCB implements to the letter, which as a whole has been proven correct and complete up to equivalent classes. □

[1] I would like to thank Dr. Denver Dash for providing me with enormous support for this proof

Algorithm 22 Main controller for PCB

Require: Completely connected, undirected graph G,

max conditioning set limit i_{max},

maximum number of Mapper jobs per MapReduce job m_{max}

1: **for** $i = 0$; $i < i_{max}; i++$ **do**

2: Count number of edges in G: $n = |\mathbf{E}(G)|$

3: Calculate how many edges should assigned to each Mapper job:

$K = \lceil \frac{n}{m_{max}} \rceil$

4: Readjust number of Mapper jobs:

$m = \lceil \frac{n}{K} \rceil$

5: Upload G to the cluster

6: construct input file I for Mappers that contains an edge starting position s_i for each of the m Mapper jobs:

7: **for** $z = 0; z < m; z = z + K$ **do**

8: Write z to file I

9: **end for**

10: Start the MapReduce job with parameters i, n, m, K (Algorithms 20 and 21)

11: Download the output list ($\mathbf{L}$) from the cluster (Algorithm 21)

12: Decode the list and update the graph:

13: **for** each $((X, Y), \mathbf{C})$ in the file **do**

14: remove edges (X, Y) and (Y, X) from G

15: Store $\mathbf{C}$ for later processing in the edge orientation step of the PC algorithm [Spirtes et al., 2000]

16: **end for**

17: **end for**

18: Output G to the edge orientation phase of the PC algorithm

Theorem 5.4.2. *In the large sample limit, PCB is complete up to equivalence classes*

Proof. By induction, I prove that PCB finds and removes every edge (X, Y) for which X and Y are independent given some conditioning set $\mathbf{C}$.

Base Case: *Assume PCB is in phase $i = 0$ and has a completely connected graph G.*

PCB will test all pairs of variables X and Y for marginal independence (conditioning set size $i = 0$), and regardless of how the edges are distributed over partitions, it will correctly find all marginal independence statements $I(X, Y)$. The result of each marginal independence tests does not influence any of the other tests, and thus they can be determined in any order and distributed over any configuration of partitions without consequences.

Induction Step: *Assume PCB is in phase i and currently has a graph G, which has had all edges (X, Y) for which we have found $I(X, Y \,|\, \boldsymbol{C})$, for $|\boldsymbol{C}| < i$, removed. Then, for phase $i + 1$, in any partition, testing any arbitrary edge (X, Y) and for which there exists a set $\boldsymbol{C}$ such that $I(X, Y \,|\, \boldsymbol{C})$ and $|\boldsymbol{C}| = i + 1$, then this edge (X, Y) will be removed by PCB.*

From the assumption of the induction step we know that X and Y need at least $i + 1$ variables to d-separate them. There must exist a set $\mathbf{C}$ of $i + 1$ variables such that $c \in \mathbf{C}$ and $c \in \mathbf{Adj}(X)$ [proof by Spirtes et al., 1993, is valid for Y as well].

Since at phase $i + 1$, in the worst case, PCB will check all $\binom{|\mathbf{Adj}(X)|}{i+1}$ combinations of $\mathbf{Adj}(X)$, and it will eventually query the oracle for $I(X, Y \,|\, \mathbf{C})$, resulting in the removal of the edge from G, assuming PCB did not already find another conditioning set of size $i + 1$ capable of d-separating X and Y. □

5.5 EMPIRICAL EVALUATION

As stated in the introduction, my aim for developing MapReduce versions of the PC Algorithm was to exploit the constraint-based approach to learn network structures from data with many variables. My target was to be able to handle datasets with a total number of variables up to 5,000-10,000, having not necessarily many records. The two reference algorithms [Chen et al., 2011, Fang et al., 2013] focus more on the number of records than the number of variables. This has influenced their algorithm design where their Mapper and Reducer

functions focus almost completely on counting and are not very complex. Consequently the MapReduce jobs are called frequently in commonly used subroutines of the algorithms. For each mutual information calculation [Chen et al., 2011] or each Bayesian score calculation [Fang et al., 2013] a MapReduce job will have to be executed on the cluster. Since each MapReduce job means, from a computational point of view, a significant time overhead, running many of them may prohibit these algorithms from scaling to datasets with many variables.

The MapReduce jobs used by my algorithms, PCA and PCB, are based around the iterations of independence tests of the PC algorithm, where each iteration applies independence tests with a larger conditioning set size than the previous one. When we limit the conditioning set size, typically to eight conditioning variables, my algorithms roughly require a constant number of MapReduce jobs to complete the work. The consequence of this decision is that the Mapper and Reducer functions are more complex and require more from the hardware used for communicating data and performing the calculations.

My empirical evaluation consists of two parts, a gold standard recovery task and a classification task. Both pit the four algorithms against each other and a single computer implementation of the PC algorithm from the SMILE library as a reference. Both tasks involve a number of datasets that are either generated from well-known Bayesian networks from the literature, or are well-known datasets available in the UCI data repository.

5.5.1 Computing Environment

The Hadoop algorithms, written in Java and using a few native C++ compiled parts from SMILE, were run on a cluster with 20 high memory computers. Each computer, or node, has 16 2GHz CPU cores, 382 GB RAM and 8 300GB Hard Drives. The Mapper and Reducer code were executed in a Java virtual machine (VM) with a maximum of 4GB of memory allocated to them. The controller code was not run on the cluster, but within a virtual machine, running Debian Linux. The virtual machine has 8GB of memory and has 2 CPU cores assigned to it. The VM ran on a laptop with a 2.2GHz quad core processor with 32GB RAM and a 512GB SSD. SMILE was run on a computer running Ubuntu Linux with

a 2.4GHz dual core processor, 8GB of RAM and 2 1.5 TB hard drives in a RAID mirror configuration. The PC algorithm was constrained to use up to 4GB of memory. The SMILE program was written in C++ and compiled to run natively on the Linux platform for optimal performance.

5.5.2 Experiment 5.1: Gold Standard Recovery

5.5.2.1 Methodology The first part of the empirical evaluation focuses on the ability of the algorithms to recover a gold standard structure from data. For this task I have selected eight different networks to serve as gold standards. They are listed in Table 5.1, ordered by size.

Table 5.1: Gold Standard Evaluation Networks used for Experiment 5.1

Network	Nodes	Edges	Reference
Alarm	37	46	Beinlich et al. [1989]
Hailfinder	56	66	Abramson et al. [1996]
Hepar	70	123	Onisko [2003]
CPCS179	179	239	Pradhan et al. [1994]
Andes	209	388	Conati et al. [1997]
Alarm10	370	570	Tsamardinos et al. [2006]
Hailfinder10	560	1017	Tsamardinos et al. [2006]
Gene	801	972	Tsamardinos et al. [2006]

Initial tests showed most algorithms would take an unacceptable amount of time when datasets had more than 1,000 variables, thus datasets where chosen with fewer variables to allow for sufficient datapoints in the analysis.

The four different Hadoop algorithms and the SMILE baseline are evaluated on the following criteria:

1. Running Time (ms)
2. Hamming Distance
3. Number of MapReduce jobs executed by the algorithm
4. How close to finishing was the algorithm when it was terminated.

The final criterion is necessary due to external circumstances. To run any of the MR algorithms, there must be an open connection with the Hadoop cluster. This cluster is located in a remote location and I did not have complete control over the accessibility to the cluster and its maintenance schedule. Running the algorithms, which could potentially take multiple days to weeks to finish, in this environment could result into algorithm termination due to connection loss, (un)scheduled maintenance, and other unexpected events. I ran the algorithms during the day, while at work and during the night, while at home, allowing them a total of around 9-12 hours of running time. The progress of all the algorithms is quantifiable, for instance, by looking at how many phases they have completed [Chen et al., 2011], or how many parent sets have been finalized [Fang et al., 2013].

5.5.2.2 Results The results for the evaluation are listed in Table 5.2. For each algorithm I list their gold standard recovery performance for the different networks. With the exception of two network results from the PCA algorithm, running time results are reported for all algorithm / network combinations. A significant portion of the networks could not be finished by the Chen and Fang algorithms. This is reported by a '-' in the Hamming Distance column and by a number less than 100 in the Completion column. Running times for these cases are approximate and are based on the timestamps from the Hadoop log printout, which do not allow for millisecond precision and were typically rounded to the nearest minute. Figure 5.1 shows the algorithm performance in respect to the Hamming Distance. In general, all algorithms perform similarly. Fang's algorithm performs better than the other algorithms on the network structures it was able to finish, but it benefited from having a correct node ordering provided to it. Chen's algorithm performed markedly worse on the Alarm network,

Table 5.2: Gold Standard Evaluation Results of Experiment 5.1

Chen	Running Time (ms)	Hamming Distance	MR Jobs	Completion (%)
Alarm	7,946,158	91	255	100
Hailfinder	28,800,000	-	132	61.5
Hepar	1,175,410	151	30	100
CPCS179	40,800,000	-	670	66.2
Andes	12,028,230	522	390	100
Alarm10	39,600,000	-	833	60.3
Hailfinder10	37,080,000	-	888	48.9
Gene	32,400,000	-	520	35.6
Fang	Running Time	Hamming Distance	MR Jobs	Completion (%)
Alarm	5,836,158	3	203	100
Hailfinder	8,677,891	39	308	100
Hepar	12,161,432	45	412	100
CPCS179	33,480,000	-	968	80.4
Andes	43,800,000	-	1251	87.5
Alarm10	43,800,000	-	1208	41.4
Hailfinder10	33,480,000	-	868	25.9
Gene	43,080,000	-	1087	22.2
PCA	Running Time	Hamming Distance	MR Jobs	Completion (%)
Alarm	898,916	22	24	100
Hailfinder	921,367	60	24	100
Hepar	985,596	116	24	100
CPCS179	2,544,909	264	24	100
Andes	740,000	-	4	15
Alarm10	1,598,000	-	4	20
Hailfinder10	-	-	-	-
Gene	-	-	-	-
PCB	Running Time	Hamming Distance	MR Jobs	Completion (%)
Alarm	241,874	27	8	100
Hailfinder	355,508	61	8	100
Hepar	264,795	152	8	100
CPCS179	432,923	297	9	100
Andes	328,082	503	8	100
Alarm10	437,592	953	8	100
Hailfinder10	502,944	1256	8	100
Gene	572,099	1992	5	100
SMILE	Running Time	Hamming Distance	MR Jobs	Completion (%)
Alarm	786	19	-	100
Hailfinder	1,786	69	-	100
Hepar	1,816	153	-	100
CPCS179	36,883	298	-	100
Andes	15,371	518	-	100
Alarm10	50,454	966	-	100
Hailfinder10	167,827	1309	-	100
Gene	1,572,861	1998	-	100

but this seems to be an anomaly since it performs similar to the other, PC algorithm based, algorithms on the other network recovery tasks it was able to finish. The three algorithms based on the PC algorithm, PCA, PCB and SMILE's PC algorithm implementation all performed similar, showing that PCA and PCB are at least valid alternatives to SMILE's PC algorithm.

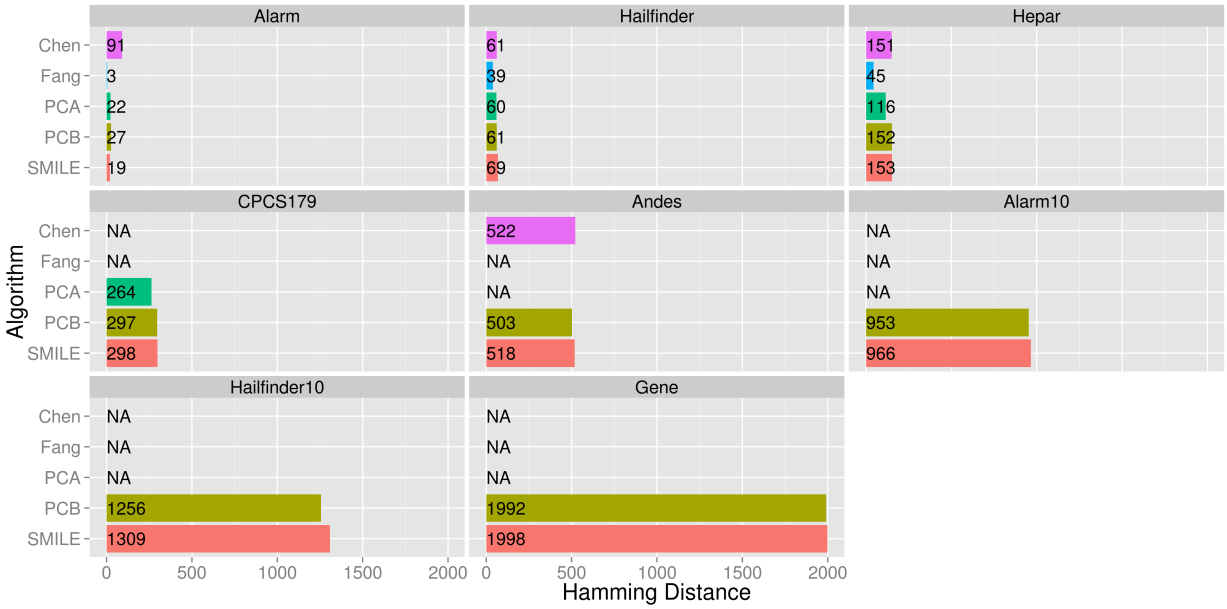


Figure 5.1: Hamming Distances for Gold Standard Recovery Task

Comparing the MapReduce algorithms, the number of MapReduce jobs that each of the algorithms required for the work is markedly different. As was discussed in Section 5.5, mainly the difference is due to different design decisions made. Figure 5.2 shows the number of MapReduce jobs used by the algorithms. The PCA and PCB algorithms ran a near constant number of MapReduce jobs, dependent on how many different conditioning set sizes were tested. The number of MapReduce jobs varied more for Chen's and Fang's algorithm. Typically, more were needed when the network to be recovered had more variables, but exceptions did occur. Most notably, the Chen algorithm required only 30 MR jobs when it recovered the Hepar network, much less than for any of the others. For the Alarm network it even required 255 jobs. The number of jobs seems to be quite data dependent.

The structure of the Fang algorithm clearly shows that an increase in the number of variables in the dataset will increase the number of MapReduce jobs that will have to be run. For each variable we run at least two MapReduce jobs. One to obtain the parentless baseline score, and one job to determine a potential parent for the node in the network. If adding a parent increases the score, the process continues and the algorithm will try to add

more parents. There might be some variance in the total number depending on the data. For the three networks the Fang algorithm was able to successfully recover a network structure, I found that the number increases with the number of variables involved.

For the other, failed attempts there is some variance in the number of MapReduce jobs. This is partially due to a variation in total running time, but might also be due to the different datasets.

Figure 5.2: Number of MapReduce Jobs Used When Recovering Gold Standard Networks

Having established that all algorithms have similar performance for the gold standard recovery task, we can examine the running time of the algorithms. Figure 5.3 shows the algorithm running time for each of the network recovery tasks.

This figure paints a disheartening picture: for most of the networks to be recovered in the experiment, SMILE's PC algorithm implementation is, vastly superior to any of the Hadoop-based algorithms. Each MapReduce job that is run on the cluster takes at the very least an amount of time in the order of seconds. In this amount of time, SMILE is able to finish most of the tasks, being at least as accurate, but much faster. Granted, both Chen's and Fang's algorithm are aimed more at datasets that are large in the number of records,

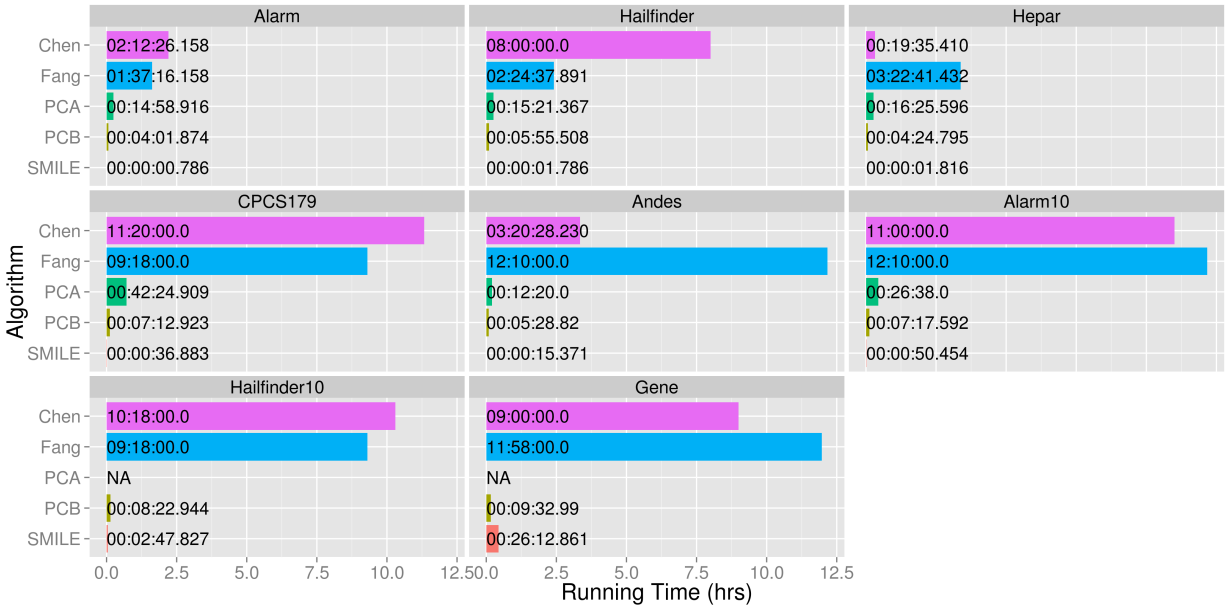


Figure 5.3: Algorithm Running Time During Gold Standard Recovery Task

but taking into account all the optimizations that are present in SMILE's PC algorithm, I believe as long as there is sufficient memory available to the algorithm, it will continue to beat the other algorithms until the datasets become sufficiently large. Furthermore, as Figure 5.2 showed, if a dataset under investigation has a larger number of variables, Chen and Fang's algorithms may end up taking an enormous amount of time due to the sheer number of MapReduce jobs they need to run. Section 5.5.3 shows one example dataset that contains thousands of variables, in this case even SMILE and the PCB algorithm take an enormous amount of time to finish. This is mainly due to the sheer number of edges that require testing.

Figure 5.3 shows at least one positive point. When the number of variables starts to increase, the total running time required by SMILE does increase as well, and at some point even surpasses the running required by the PCB algorithm. Section 5.5.3 shows even more examples where the PCB algorithm finished the classification model learning task quicker than SMILE. Initially, SMILE benefits from performing all the calculations in memory without the need for network communication and the added complexity required for dividing up

the work. SMILE was written in C++ and is compiled natively for the Linux machine used for running the experiments. These are all benefits SMILE has over the Java based implementation of the PC algorithm used within the Hadoop framework. Java code is executed within a virtual machine, which has many benefits as well, the ability of executing Java code on many platforms being one of them. A downside is that code is executed markedly slower. Anecdotal evidence, which I acquired by running the Java version of the PC algorithm that is present within the SMILEWIDE library, suggests that the difference is an order of magnitude, probably around 7-10 times slower than SMILE's C++ implementation.

Consequently, evidence suggests that before the MapReduce based PCB algorithm manages to surpass SMILE's single processor performance, we will require datasets that have at least hundreds of variables, and need to have access to a cluster that allows for a large number of Mapper tasks to be run in parallel.

5.5.3 Experiment 5.2: Classification

The second part of the empirical evaluation is a classification task. Using datasets from the UCI Machine Learning Repository and the Causality Workbench,[2] we learn classification models using the algorithms and compare their performance on classifying samples for a test set.

After running the gold standard recovery task, I found that some of the algorithms would either take a very long time to finish (Chen, Fang), or would not finish at all (PCA). This influenced my choice of datasets for the classification task. In total I chose four different datasets, two where quite small, having only 22 and 37 variables respectively, and two that were larger. The larger two had 257 and 4,933 variables. These were chosen to investigate the performance of the PCB algorithm and the SMILE implementation of the PC algorithm. Both of these were able to finish much quicker than the other algorithms and most likely would be the only two that would be capable of handling datasets of such size in a reasonable amount of time.

[2]The Causality Workbench can be found at http://www.causality.inf.ethz.ch/repository.php

5.5.3.1 Methodology For the classification task four datasets were chosen, three from the UCI Machine Learning Repository and one from the Causality Workbench. Table 5.3 shows the datasets chosen. The four algorithms were used to learn Bayesian network structures (Fang), or equivalence classes (the 3 others). Using a conversion algorithm implemented in SMILE, which includes the use of the algorithm by Dor and Tarsi [1992], I converted the equivalence classes into proper DAGs. After acquiring the BN structures I, running the EM algorithm on a single pc, learned parameters for the networks using the training data. I used the test data to classify samples. To summarize, the experiment procedure consisted of the following steps:

1. Divide up the data in a training set (90%) and a test set (10%).
2. Learn model structure using the training data.
3. If the structure is not a DAG, convert to DAG.
4. Learn parameters for the structure using the training data.
5. Evaluate the model performance by using the test data.
6. Record performance metrics:
    - Running Time
    - Classification Accuracy
    - Number of Map Reduce Jobs
    - Completion Percentage

Table 5.3: Classification Datasets used for Experiment 5.2

Datasets	Variables	Records	Classes	Reference
Mushroom	22	8,124	3	Bache and Lichman [2013]
Chess	37	3,196	2	Bache and Lichman [2013]
Semeion	257	1,593	10	Bache and Lichman [2013]
Sido	4,933	12,678	2	Causality Workbench Team [2008]

5.5.3.2 Results As was expected, the algorithms by Chen and Fang only completed the two smaller datasets in a reasonable amount of time. The PCA algorithm managed to complete the first two datasets as well, but could not finish the Semeion dataset. Due to its sheer number of variables, I did not attempt to run any of these algorithm on the Sido dataset.

The PCB algorithm and the SMILE implementation both were able to finish the first 3 datasets within a reasonable amount of time, but the Sido dataset ended up taking too long. The PCB algorithm was terminated after 24 hrs, and the SMILE implementation ended up running for another 14 hours. Both were still only testing conditional independence with just one conditioning variable. The PCB algorithm had managed to finish 29% of the edges, while the SMILE algorithm, only finished about 3% of the edges, even though it ran for an additional 14 hours. It may have taken weeks to completely finish learning the models, especially since the number of independence tests potentially increases exponentially with the size of the conditioning set. The completion percentage listed for the PCB and SMILE algorithm is deceptive, since it does not account for the increasing number of tests to be performed per edge, and only accounts for how many edges were completed per stage and how many stages were completed.

Figure 5.4 show the classification results of the algorithms. The red line in the graphs shows a baseline classification accuracy, that is obtained by simply determining the base rates of the class variable from the test data and selecting the most likely class instance as the default answer for this "classifier" when classifying the test samples. Figure 5.5 shows the running time of the algorithms. No figure has been included for the number of MapReduce jobs used by the algorithms for the classification task, as they follow the same pattern as in Experiment 5.1. The numbers are available in the table.

One notable observation we can make from the results of Experiment 5.2 is that, in contrast to the gold standard recovery task, the PCB algorithm does not seem to perform as well as the other algorithms. In all cases where the algorithms were able to finish the task, SMILE is consistently either the best or second best algorithm. There was only one case where Chen's algorithm was able to finish the task completely. It did manage to learn a structure for the Mushroom problem, but the network pattern structure could not be

Table 5.4: Classification Task Results of Experiment 5.2

Chen	Running Time (ms)	Accuracy (%)	MR Jobs	Completion (%)
Mushroom	22,930,689	-	222	100
Chess	3,689,175	13.7	122	100
Semeion	28,800,000	-	807	37.7
Sido	-	-	-	-
Fang	Running Time (ms)	Accuracy (%)	MR Jobs	Completion (%)
Mushroom	7,214,637	35.7	260	100
Chess	12,533,442	53.7	425	100
Semeion	36,000,000	-	982	56.4
Sido	-	-	-	-
PCA	Running Time (ms)	Accuracy (%)	MR Jobs	Completion (%)
Mushroom	1,124,029	37.8	24	100
Chess	1,172,064	30.3	24	100
Semeion	3,600,000	-	7	35
Sido	-	-	-	-
PCB	Running Time (ms)	Accuracy (%)	MR Jobs	Completion (%)
Mushroom	213,175	37.8	9	100
Chess	266,824	25.8	9	100
Semeion	1,991,374	2.5	7	100
Sido	86,400,000	-	2	0.143
SMILE	Running Time (ms)	Accuracy (%)	MR Jobs	Completion (%)
Mushroom	11,102	41.0	-	100
Chess	15,564	40.7	-	100
Semeion	8,431,207	31.5	-	100
Sido	139,216,695	-	-	0.115

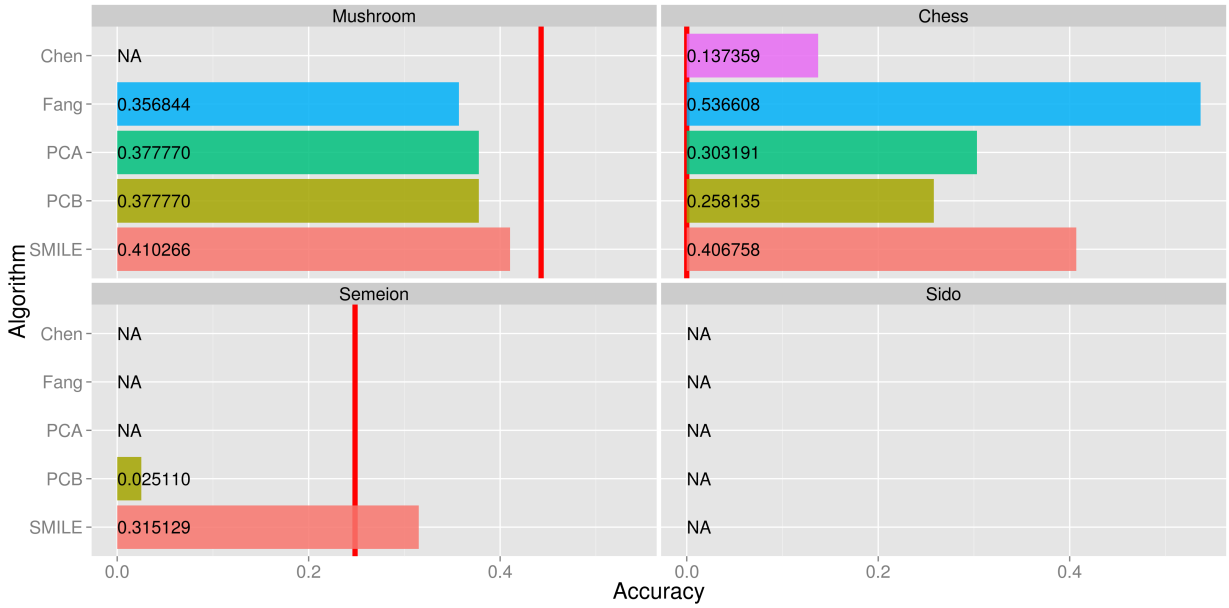


Figure 5.4: Classification Accuracy Results of Experiment 5.2

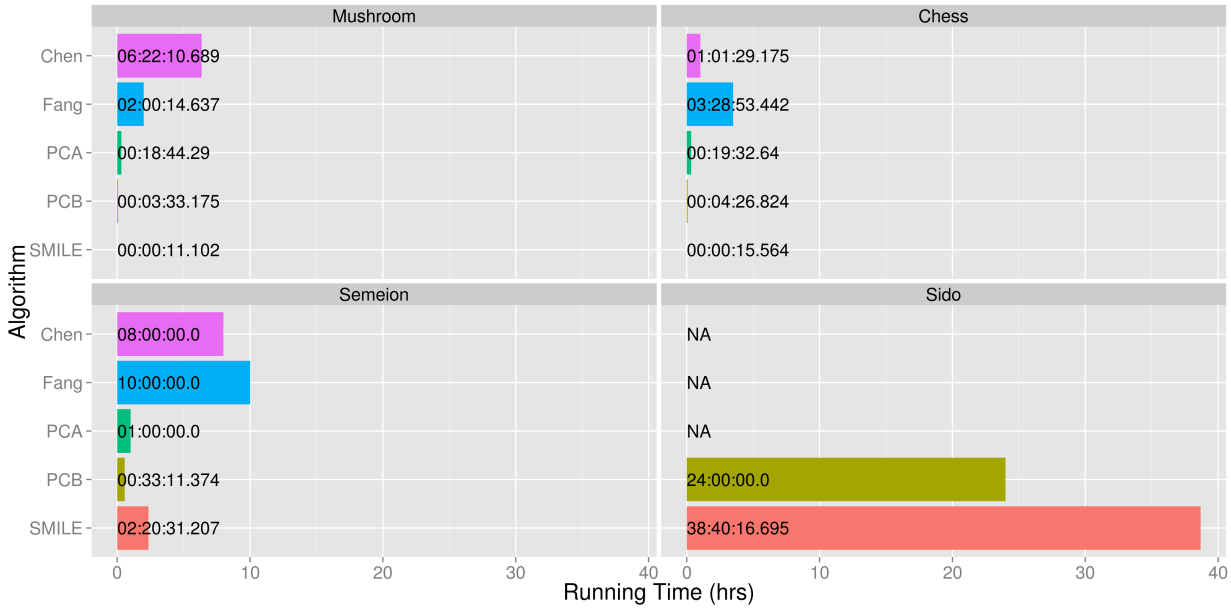


Figure 5.5: Algorithm Running Time During Classification Task

translated into a Bayesian network without memory problems due to very large parent sets for some of the nodes. PCA has mixed results, performing poorly on the Mushroom data and quite well on the Chess data.

All algorithms performed poorly on the Mushroom data. Partially we can blame the methodology. Effectively, we performed a 1-fold cross validation, where we divided the data in one training set and one test set. An unlucky choice of test samples may have caused the models to perform worse than they may have done if other samples would have ended up in the test set. Given that running a more extensive k-fold cross validation would be very time consuming, I performed the procedure only for the three PC algorithm implementations and left out the large Sido dataset. The results are shown in Figures 5.6 and 5.7.

Averaging the performance over five folds smoothed out the results. The PCA algorithm was not run on the Semeion dataset as it was not able to finish in the first part of Experiment 5.2. PCB and SMILE were run on all three networks, but patterns learned from Semeion (PCB) and Mushroom (SMILE) could not be converted to Bayesian networks due to large parent sets and, as a result, were left out of the evaluation. The algorithm performance on

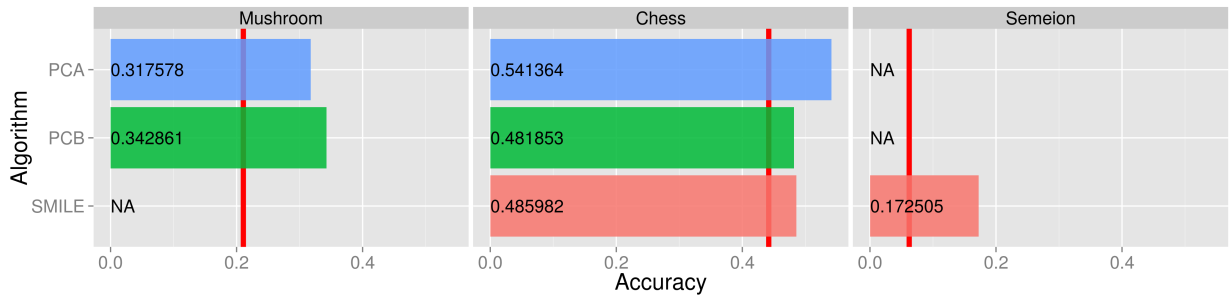


Figure 5.6: Classification Accuracy, 5-Fold Cross Validation

the Chess data now seems to be pretty equal. For the other two datasets the performance has decreased compared to the single-fold result, ending up probably closer to the real performance of the algorithms.

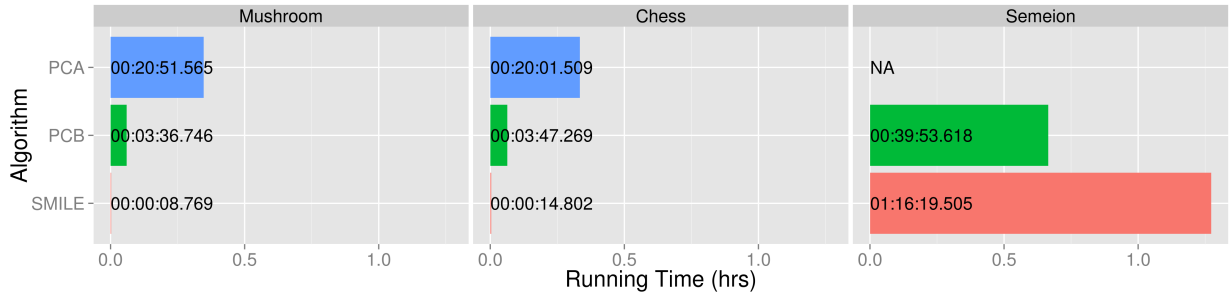


Figure 5.7: Running Time, 5-Fold Cross Validation

A possible explanation for the accuracy differences between PCA, PCB, and SMILE, which all implement the PC algorithm, is that there are minor differences in the implementations that could have a big impact on the end result. SMILE performs all the independence tests sequentially. The benefit of the sequential approach is that any new information obtained from the independence tests is available to all tests that are performed afterwards. Any removed edge will exclude variables from conditioning sets in future tests, which has a potential impact on future contingency tables used by the PC algorithm for independence tests. If a variable with many states is no longer present in conditioning sets, especially in

the later stages, this will result in significantly smaller contingency tables. Comparing PCB and SMILE more closely, we can hypothesize why PCB performs worse than SMILE. Since PCB partitions the edges to allow multiple independence tests are performed in parallel, the partitions perform these independence tests with not the most up to date information of the network structure. It is very likely that the PCB algorithm is performing independence tests using conditioning sets that contain variables that would not have been included if the information on all edges that were deemed independent was available to all Mappers at all time. This may cause the contingency tables to become sufficiently large that the required samples-per-cells ratio recommended by Tsamardinos et al. [2006] can no longer be satisfied, resulting in the edge to be removed. To test this explanation, I reran the PCB algorithm on the 1-Fold Mushroom and Chess data. I varied the maximum number of Mapper jobs the PCB algorithm would create for each MapReduce phase. This setting influences how the edges that need testing are partitioned. Edges are divided equally over the Mapper jobs. If the Mapper setting is bigger than the total number of remaining edges, the number of Mapper jobs is pruned in such a way that all Mapper jobs contain the same number of edges. In the extreme case all Mappers will contain only one edge. The Mapper jobs are divided up by the Hadoop cluster over the computation nodes. The cluster I had access to was able to process about 230 Mapper jobs in parallel. Figures 5.8 and 5.9 show the results of this test. I set the maximum number of Mapper jobs to 5, 10, 50, 100, 500, and 1000. Figure 5.8 shows the impact of changing the maximum number of Mapper jobs the algorithm is allowed to create. The red line represents the classification accuracy of SMILE's PC algorithm on

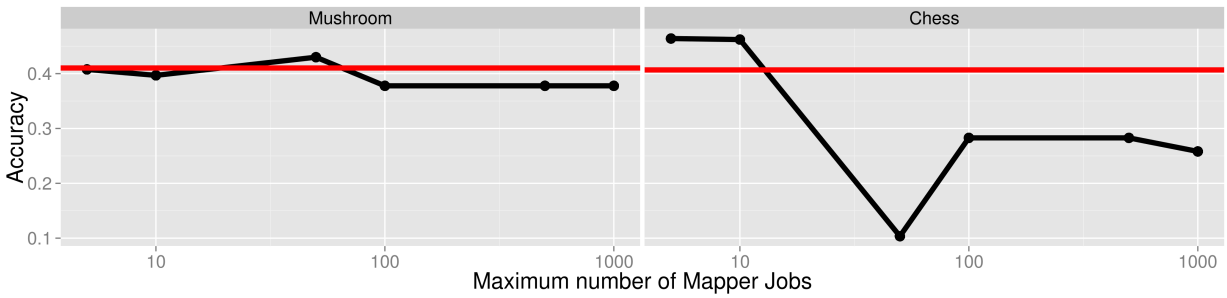


Figure 5.8: Classification Accuracy, Varying the Maximum Number of Mapper Jobs

the respective dataset. Figure 5.9 shows the results of the PCB algorithm with the different Maximum Mapper settings.

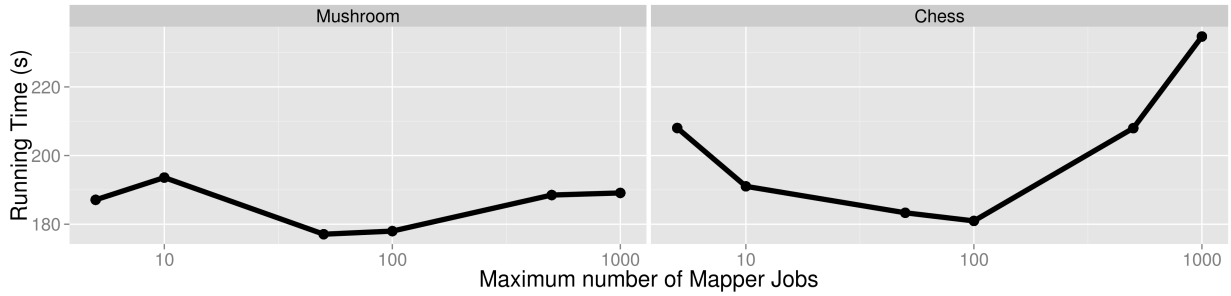


Figure 5.9: Running Time, Varying the Maximum Number of Mapper Jobs

The differences are not very apparent in the Mushroom dataset, but this might be because it is the smaller of the two. In the chess dataset, increasing the maximum number of Mapper jobs available to the algorithm seems to have a negative impact on the classification accuracy of the model. With only 5 or 10 Mapper jobs to divided the edges over, the PCB algorithm outperforms SMILE, beyond that it underperformed with a very dramatic drop in performance when the maximum was set to 50. Comparing the best and the worst network (Figures 5.10 and 5.11), we see that there is a significant difference between the Markov blankets of the class variables (class variable and Markov blanket members are positioned top left). Only three variables occur in both Markov blankets, and the number of direct adjacent variables differs significantly between the two networks. In the better performing network (Fig. 5.10), the class variable is connected to 7 variables, in the other, only to 4. The only difference between the two algorithms runs is the maximum number of Mappers and consequently, how the edges were partitioned.

Figure 5.9 shows, that if too many Mapper jobs are created, the overhead of the Hadoop system starts to slow down the process. With more Mappers jobs, there is more communication necessary on the cluster. Every Mapper job is executed by a Mapper instance, which needs to read the data, initialize, execute the Mapper function, and write the data. This data then needs to be send to Reducer instances, etc. We have to conclude that a careful

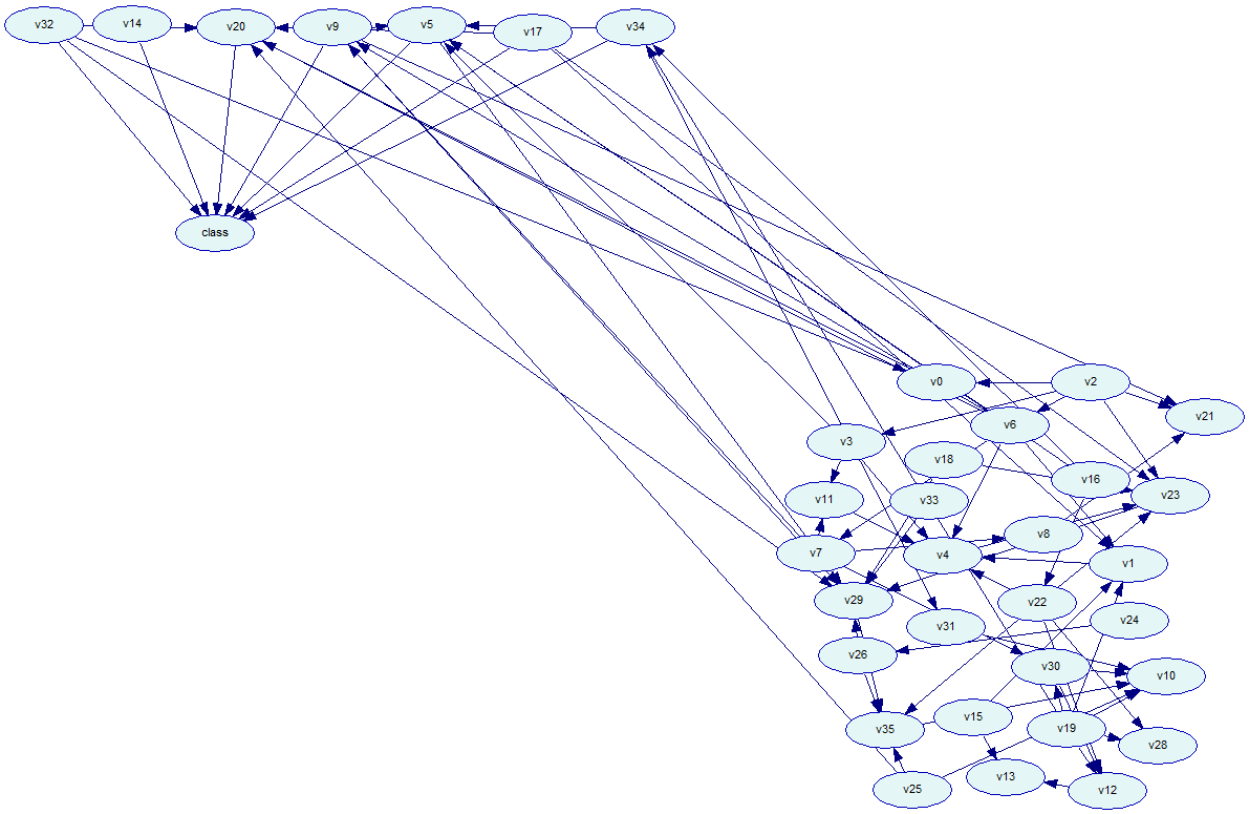


Figure 5.10: Chess network, Maximum Number of Mappers: 5

consideration of the edge partitioning is necessary. Utilizing too many mappers, although facilitating a larger scale of parallelization, might lead to suboptimal performance.

5.6 DISCUSSION

The idea of applying a MapReduce cluster to learning Bayesian network structures is an attractive one. The complexity of BN structure learning algorithms ensures that when the data becomes less trivial, i.e., by having more records or/and more variables, the time required to learn models will quickly increase beyond reasonable limits. Moving the computations to a cluster should allow us to push the boundaries of what models we can learn in a reasonable amount of time. The MapReduce formalism, and the Hadoop implementation of MapRe-

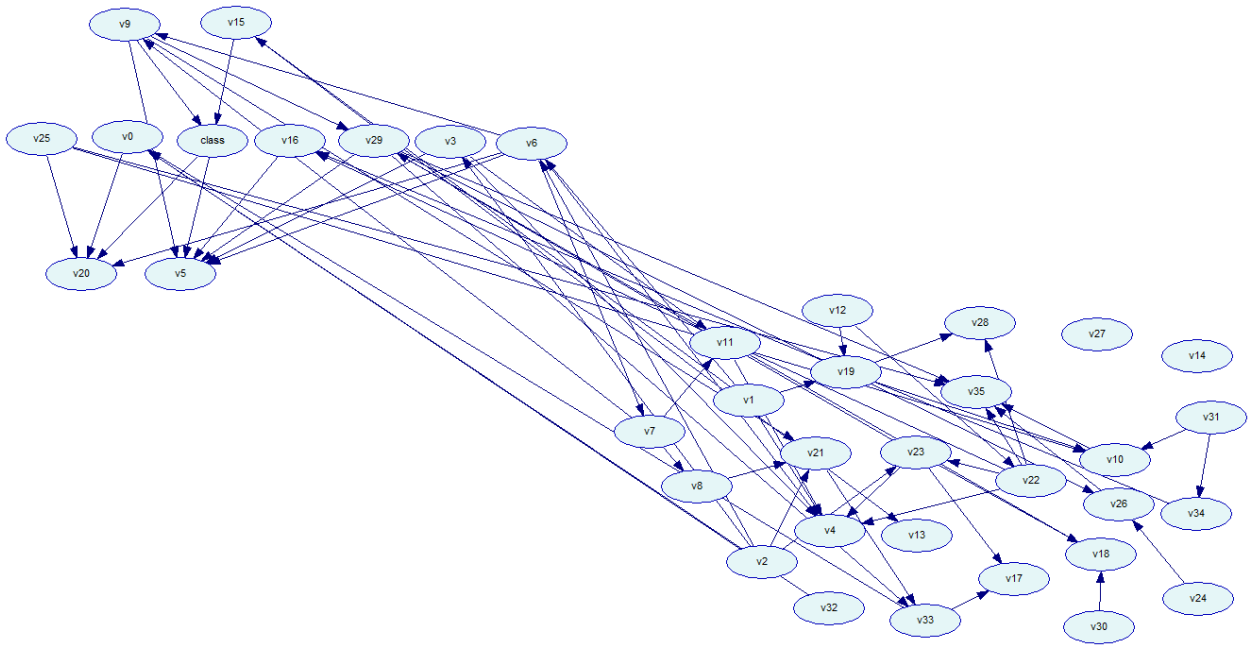


Figure 5.11: Chess network, Maximum Number of Mappers: 50

duce, are appealing candidates for the parallelization of algorithms. From a technical point of view, they offer a scalable solution that, once the algorithm has been implemented, allows for deployment on clusters ranging in size between a handful of computers and "industrial strength" clusters with thousands of computers. All this with few to no code modifications or configuration changes. However, this does not mean that the algorithm parallelization process will be obvious or easy. The MapReduce formalism defines very clearly and strictly what actions the computers in the cluster can perform. The algorithm that is to be parallelized needs to be analyzed to find parts that either naturally define Mapper and Reducer functions, or that can somehow be shoehorned into the MapReduce template.

In this chapter, I have discussed four algorithms for learning Bayesian network structures that apply the MapReduce formalism and were implemented using the Hadoop framework. Two of these were examples from the literature [Chen et al., 2011, Fang et al., 2013], the other two were proposed by me. The two-part empirical evaluation, performed on the four algorithms and a single-processor reference, showed that, under the experimental conditions,

none of the MapReduce-based algorithms yielded significant improvements in performance over SMILE's PC algorithm running on a single computer. Only specific cases may show benefits for the parallel approaches.

The two MapReduce algorithms from the literature [Chen et al., 2011, Fang et al., 2013] state a focus on the number of records. Their use of MapReduce jobs on the cluster depends on the number of variables in a dataset. Consequently, if the number of variables in a dataset increases, these two algorithm execute many MapReduce jobs on the cluster. Each of these jobs can easily take 30 seconds and upwards due to a significant amount of overhead involved with preparing the cluster to run the jobs [Cherniak et al., 2013]. Even for datasets with only 100-300 variables, both the algorithms by Chen et al. [2011] and Fang et al. [2013], routinely required more than 8-12 hours (which was the algorithm cutoff time), and their total required running time would be much longer. Neither of these algorithms seems to scale as well as was stated in the publications. One of the two [Fang et al., 2013] performed an empirical evaluation that was limited to one artificial dataset. Its data was generated using a Bayesian network with only 8 nodes. This evaluation was insufficient to provide a balanced view of the capabilities of the algorithm.

The two algorithms that were proposed by myself (PCA and PCB) had mixed success. Both would outperform the two algorithms from the literature, but would still have to yield to SMILE. Especially regarding running time, for most of the datasets, SMILE's PC algorithm was orders of magnitude faster than PCA and PCB. Only once the number of variables started to surpass a few hundred, did PCB close the gap.

Ultimately, the approach that I implemented as the PCA algorithm was not successful. Although in principle the algorithm performs correctly, in practice it cannot guarantee successful termination for every dataset. If the algorithm is able to remove many edges in the first stages (the sooner the better), then the algorithm has a good chance of finishing successfully. If this is not the case, then with almost 100% certainty the algorithm will crash.

In the remainder of this section, I will evaluate each of the four MapReduce algorithms, detailing issues and bottlenecks that occurred during development and evaluation. Finally, I will conclude this section and chapter by discussing the feasibility of applying MapReduce to the Bayesian network structure learning problem.

5.6.1 PCA

The initial tests that I ran with the PCA algorithm were promising. Network structures with a limited number of variables were learned successfully without much effort. As an example, the algorithm had no problems with learning the network structure from a 70 variable dataset with 14,000 records, which was generated using the HeparII network. Taking about 17 minutes, the algorithm was still much slower than native SMILE, which at that time would finish in about 2 minutes. Later, after improving the performance of the PCB algorithm, and having introduced these improvements into SMILE, its running time dropped to about 22 seconds.

To find out if the algorithm would scale to larger datasets as well, the next dataset under consideration was one generated using the CPCS179 network, which has 179 nodes and the generated data file in question had 100,000 records. When the algorithm was run on this dataset, its limitation had become obvious. It failed to finish, filling up the disk space of the cluster to the point where no more data could be written.

Successful termination of the PCA algorithm depends solely on its ability to eliminate edges from the graph as early in the process as possible. With many variables and a very large number of conditioning sets to consider, the number of counts that needed to be generated to perform all the necessary (conditional) independence tests became so numerous, that the output of the Mapping phase crashed the Hadoop cluster. Not enough edges were eliminated from the graph early enough to curb the number of counts that would be necessary for the conditional independence tests with larger conditioning sets.

There is a hard limit, i.e., the total amount of disk space available on the cluster, or the number of KVPs that can outputted by the Mappers and stored on the computation nodes. This implies there is a limit to even pair-wise conditional probability testing. If the number of nodes (variables) becomes too large, eventually the number of outputted KVPs will pass the hard limit. This is a potential limit of Chen's algorithm [2011] as well. In its first step, it calculates the pair-wise mutual information between all variables. All counts for this calculation are calculated in the same way as the PCA algorithm.

Without a guarantee to successfully finish learning a BN structure from a dataset, the PCA algorithm has little to no practical use. It is not able to handle datasets with many variables, and it was severely outclassed by SMILE's PC algorithm implementation for any dataset it was able to handle without crashing the cluster.

5.6.2 PCB

After it was clear that the PCA algorithm would not be able to scale to large datasets (in terms of the number of variables), I worked on an alternative that eventually became the PCB algorithm. This algorithm was successful in avoiding to crash the Hadoop cluster by overloading it with intermediate data, but it has its own issues.

To speed up the independence tests of discrete variables, PCB (and SMILE) makes use of ADTrees [Moore and Lee, 1998] to cache the necessary counts. An important issue with ADTrees is that when the conditioning set size increases, the ADTree memory usage increases exponentially, and rapidly the tree size would hit the memory limits of the Java virtual machines the Mappers are running in. This caused Mappers to run out of memory and to terminate, causing the MapReduce job to fail.

I have investigated suggestions in Moore's paper [1998] to improve the efficiency of the ADTree data structure and implemented a few of the suggestions to see if this would improve the performance of the PCB algorithm. I found that the suggestions did not help much in improving algorithm performance.

The dilemma I faced, was that the ADTree data structure is invaluable for the efficient calculation of the necessary statistics for the independence tests. If we would revert to an approach without ADTrees, i.e., iterating over the dataset to acquire counts, the process would become extremely slow, even before reaching larger conditioning set sizes.

Eventually, I found a practical workaround for the memory problems. I raised the memory limit for the Java virtual machine to 4GB and added a constraint to the ADTree data structure code that would erase the complete tree once the total memory usage of the VM hit 75%. The larger VM memory limit greatly improved the performance of the PCB algorithm, and the VM memory constraint of 75% ensured that erasing the ADTree would not

cause timeout problems for Java's garbage collector. These did occur previously when the VM memory was allowed to fill up completely.

With its memory problem under control, the PCB algorithm turned out to be the fastest of the MapReduce algorithms. It was the one capable of processing the largest datasets in terms of variables. It still requires that the whole dataset fits in memory, but this is not necessary a very debilitating constraint. Formidable datasets, worth exploring in parallel, may take just a Gigabyte of memory space, which easily fits in memory nowadays.

The empirical evaluation showed that PCB performed equally to SMILE in the gold standard recovery task, but underperformed on the classification task. Classification tasks are a special case. Typical BN algorithms used to learn classifiers, such as Naive Bayes and Tree-Augmented Naive Bayes, apply extra constraints to the BN classifier structure. Specifically, a typical constraint that is used, enforces an edge from the class variable to all other feature variables, assuming no feature selection is performed. For this evaluation, none of the algorithms did employ such a constraint and thus did not have an advantage over the others. A deeper investigation of the accuracy discrepancy between the PCB algorithm and SMILE's PC algorithm implementation provided a plausible explanation.

Evidence suggests that the partitioning process used by the PCB algorithm had a negative impact on classification accuracy in the classification task. The partitioning process, that systematically divides up edges over Mapper jobs, is mainly influenced by a parameter that controls the maximum number of Mapper jobs. More Mapper jobs means that the edges will be divided more sparsely over the jobs. The potential negative impact of edges that are spread too "thin" is that each Mapper instance that is executing a Mapper job has less up-to-date information available to it. We can illustrate this by the two extremes: 1) Allowing only one Mapper, or 2) allowing as many Mappers as there are edges, forcing the situation where each Mapper job only contains one edge. In the first case, the problem reduces to the traditional single-processor PC algorithm, where all information of removed edges is always available. In the other extreme, the Mapper instances processing the Mapper jobs will be completely ignorant of the process made by the other Mappers. Only after all conditional independence tests have been performed for a conditioning set size, the information from the individual Mappers is collected and processed in the Reducers and the main controller.

The consequence of this information ignorance, is that conditional independence tests might be performed with conditioning sets containing variables that might be still adjacent to the testing variable in this Mapper, but they may have already been declared independent in another. This results in contingency tables, used for the independence tests, to become larger than necessary. If there is insufficient data to satisfy a minimum number of samples per contingency table requirement (discussed in detail in Chapter 6), edges may be removed incorrectly. This most likely is the main cause of deviations of the result the PCB algorithm from the result of the original PC algorithm.

A small scale experiment with the PCB algorithm, varying the maximum number of Mapper jobs the algorithm was allowed to generate, showed that the classification accuracy varied significantly and, typically, decreased when the maximum number of Mapper jobs was increased.

Overall, the PCB algorithm is the best performing algorithm of the four MapReduce algorithms under consideration. Once the number of variables in the dataset grows beyond 500, the algorithm has the potential to be useful as a large-scale alternative to single-processor algorithms such as SMILE's PC algorithm. It will allow for structural analysis of datasets of that magnitude.

5.6.3 Chen's Algorithm

I implemented the algorithm by Chen et al. [2011] to use it as a frame of reference for the two algorithms I proposed. The paper contained enough detail to implement the algorithm without problems. One potential bottleneck of the algorithm is that the first MapReduce job produces the counts needed to calculate the mutual information between all pairs of variables. When the dataset has a sufficiently large number of variables, this job may crash due to Mapper message overload similar to the PCA algorithm. All the other MapReduce jobs the algorithm starts are in no danger of causing any sort of data overload on the cluster. Each subsequent MapReduce job collects counts for one conditional mutual information calculation for an edge between X and Y, and a conditioning set that contains the minimum cutset of X and Y.

In the original TPDA article [Cheng et al., 1997], the authors state that the algorithm only requires $O(N^4)$ independence tests. Since Chen's algorithm only parallelizes the calculation of the counts for mutual independence tests, we can deduce that in the worst case the algorithm will require $O(N^4)$ MapReduce jobs for all the calculations. Typically, due to overhead, a MapReduce job minimally takes at least 30 seconds and upwards, and if we run the algorithm on the scale of interest, thousands of variables, the number of MapReduce jobs that the algorithm requires may make this algorithm to become unfeasible on this scale. Effectively this algorithm is still of a serial nature, but applies parallelism for data processing.

During the evaluation, another potential bottleneck of the algorithm surfaced. The minimum cutset calculation for the conditioning sets removes the need for systematically evaluating all possible conditioning sets, which significantly improves the performance of the algorithm. I observed that, especially when the number of variables increases, the size of the cutset can become quite large. I have encountered cases where the cutset contained over 20 variables. The consequence of larger cutsets and conditioning sets is that the conditional mutual information calculation becomes more time consuming. As I mentioned in Section 5.2.3, the (conditional) mutual information measure is proportional to the G^2 statistic by a factor of $2N \log(2)$. So, in terms of a G^2 statistic calculation, we can show how the larger conditioning set impacts the calculation time. Specifically, a larger conditioning set will require a larger contingency table to be used for the calculations. We need marginal counts for each column, each row, and each "square." And these are necessary for all of the instantiations of the conditioning variables. The total number of cells increases with the size of the conditioning set and with it, the total calculation time will increase as well. How much will in part depend on the specific implementation of the MI calculation procedure. Especially with larger conditioning sets, a lot of the cells in the contingency table will end up being zero. In my implementation, only nonzero counts are collected and processed by the algorithm. This sparse representation saves a lot of space, but requires more effort when calculating the marginals, compared to a simple, but more memory-intensive table.

To summarize my experience with the MapReduce version of TPDA by Chen et al. [2011]: although the algorithm seems capable of handling datasets with many records, design choices seem to prohibit it from scaling up to datasets with many variables. This limits its use to datasets where the number of variables stays modest and the number of records is large enough to make it a viable alternative to single-processor algorithm implementations.

5.6.4 Fang's Algorithm

The algorithm by Fang et al. [2013], was the second reference algorithm I implemented for the empirical evaluation of my proposed MapReduce algorithms. The paper was not completely clear on how the authors implemented their algorithm, and as a result I had to make a few assumptions, but I believe my final implementation should be faithful to the authors' intent.

The algorithm implements the classical Bayesian search algorithm with K2 scoring in a main controller on a client computer and it outsources count calculation, Bayesian score calculation, and best candidate structure selection to the MapReduce cluster.

As with the original Bayesian search algorithm, Fang's algorithm assumes a node ordering. This allows the algorithm to optimize the parent set of each node independently. The authors do not discuss in their paper how one should acquire a good node ordering, which is a severe barrier to applying this algorithm to large scale datasets with many variables.

For the empirical evaluation I provided node ordering in the following ways: 1) for the gold standard recovery task I provided the algorithm with a correct node ordering derived from the gold standard network 2) for the classification task I provided a fixed node ordering and this ordering was used for all experiments.

Similar to the algorithm by Chen et al. [2011], Fang's Bayesian search algorithm requires many MapReduce jobs to complete the search for the optimal BN structure. The algorithm needs to run at least $3N$ MapReduce jobs on the cluster, with a worst case complexity of $O\left(N^2\right)$ MapReduce jobs.

The maximum number of parents of a node has been limited to 8, which helps to curb the total running time of the algorithm. It will also, barring cases with nodes with many variable states, help to ensure that the number of parameters necessary for the Bayesian

network stays reasonable. Additionally, it will keep the score calculation tractable, avoiding the bottleneck that is present in Chen's algorithm.

Overall, in the empirical evaluation I found that the algorithm typically required a large amount of time to complete the tasks, but, helped by the provided node orderings, it performed adequately. However, taking into account the reliance on many small MapReduce jobs, and the absence of a mechanism to determine a valid node ordering from the provided data, I believe the usefulness of this algorithm, similarly to Chen's algorithm, is limited to datasets with smaller numbers of variables and a large number of records. Outside of this domain it is outclassed by single-processor algorithms such as those present in SMILE, or the PCB algorithm.

5.6.5 The Feasibility of MapReduce Algorithms for Learning the Structure of Bayesian Networks

Having discussed the four algorithms separately, I will conclude this chapter with a general discussion on the feasibility of applying the MapReduce framework to solving the Bayesian network structure learning problem.

With the ever increasing amount of data available in the world, we have arrived in an era where we have so much data available to us that it seems we continuously have to find new places to put it, let alone have sufficient resources to make sense of it all. Most machine learning techniques, including the Bayesian network formalism, rely on algorithms that are exponential in time or space. Naturally, with the increased availability of parallel computation platforms, parallelized versions of machine learning algorithm (including Bayesian network algorithms) started to appear. Where in the early days the objective of these parallel platforms was more focused on dividing up complex computational work, with the introduction of the MapReduce formalism, the focus shifted more to improving the ability of processing massive amounts of data.

The ability of MapReduce platforms such as Hadoop to scale up data processing tasks to an enormous magnitude, caught the eye of many researchers, including those in the area of Bayesian networks. We can find examples of Bayesian network inference [Ma et al., 2012]

and parameter learning [Basak et al., 2012]. Mapreduce algorithms for BN structure learning were proposed by Chen et al. [2011] and Fang et al. [2013].

My work focused on structure learning and my aim for this work was to propose algorithms that would be able to handle datasets with thousands of variables. The current state of the art, work by Chen et al. [2011] and Fang et al. [2013], was more focused on datasets with large numbers of records than large numbers of variables, and I believed that their algorithm would not scale well in this area. Having proposed two MapReduce algorithm based on the PC algorithms, I performed an empirical evaluation that compared the performance of the four MapReduce algorithms on two separate tasks. Additionally, I compared the performance of the MapReduce algorithms with the performance of an algorithm run on a single-processor computer (SMILE's PC algorithm).

I found that, in most cases covered in my empirical evaluation, all four MapReduce algorithm were severely outclassed by the single-processor algorithm in terms of running time, Hamming distance, and classification accuracy. SMILE's PC algorithm implementation would perform as well or better, in a fraction of the time required by the other algorithms. Only when the number of variables started to increase beyond 500 variables, did the PCB algorithm catch up to, or surpass the performance of the regular, single-processor PC algorithm.

From the results of the empirical evaluation, I have to conclude that it seems that there are only specific uses for MapReduce-based algorithms for Bayesian network structure learning. In most cases running a algorithm on a single computer will not only suffice, but will actually be the optimal choice. For the datasets and tasks that were part of my evaluation I found that running an algorithm on a single-processor platform will give you similar or better results in a much shorter time, without the need for a cluster of computers. The main bottleneck of the continued success of algorithms such as SMILE's PC algorithm will be the amount of internal memory available to it. Using advanced data structures such as ADTrees allows the algorithm to trade time for space, and with sufficient space, we will be able to process datasets of increasing size on single computers.

The best case for a MapReduce algorithm seems to be the PCB algorithm, where we distribute the conditional independence testing over many processors. PCB was designed

for the case of datasets with many variables and typically a more restricted number of records. The independence tests are the premier bottleneck of the PC algorithm, and, when the number of dataset variables increases, the sheer number of tests provides compelling evidence for a parallel approach. As the second part of the empirical evaluation showed, the simple partitioning scheme currently used by PCB does not provide the algorithm with optimal results. A potential solution might involve creating overlapping partitions. This means doing more work than necessary. But, if the duplicate edges are chosen wisely, this may result in more optimal series of independence tests, allowing the algorithm to produce a better, final Bayesian network structure.

6.0 EVALUATION OF RULES FOR COPING WITH INSUFFICIENT DATA

This chapter describes two experiments that investigate a discrepancy in the literature on how to handle the case of insufficient data when performing independence tests for the PC algorithm.

6.1 INTRODUCTION

If we examine the PC algorithm [Spirtes et al., 1993] more closely, we find that the most computationally intensive part of the algorithm is the first phase. In this phase we perform (conditional) independence tests to determine the skeleton of the Bayesian network. For an edge (X, Y) under consideration, with or without conditioning set $\mathbf{S}$, we construct a contingency table. Using this table, we calculate a test statistic, typically the G^2 statistic, and, using a χ^2 distribution, we determine a p-value which we compare against our preset significance level α, finally removing the edge if the p-value is larger than α.

The quality of a Bayesian network produced by the PC algorithm depends mostly on the first phase, as it will deliver the skeleton of the network, which remains unchanged afterwards. For each edge deemed independent, we collect the conditioning set $\mathbf{S}$, which also influences the result of the edge orientation phase.

In turn, the quality of the first phase of the PC algorithm depends on the quality of the performed independence tests. If there are insufficient data available for the tests, we may end up making flawed statistical decisions.

In their book, Spirtes et al. [1993] discuss a minimum ratio of 10 to 1 of samples to contingency table cells. This ensures a minimum level of reliability for the statistical tests. The advice is, that when the ratio is not satisfied, the test should not be performed and the variables of the edge should be considered dependent.

When there are insufficient data available, we find that the contingency tables will have many zeros, i.e. we cannot find co-occurrences of the variable/state assignments. This will reduce the value of the G^2 statistic and the number of degrees of freedom used for the χ^2 test. A smaller value for G^2 can make it seem that the two variables of the edge are independent of each other. Consequently, independence tests for which we have insufficient data available can result in erroneous statistical decisions of independence.

I have come across a statement by Tsamardinos et al. [2006], where they state that they follow the advice by Spirtes et al. [2000], but where Spirtes et al. recommend keeping an edge if there are insufficient statistics, Tsamardinos et al. recommend removing the edge. To be exact, Spirtes et al. [2000] state:

> *In testing the conditional independence of two variables given a set of other variables, if the sample size is less than ten times the number of cells to be fitted we assume the variables are conditionally dependent.*

Curiously, Tsamardinos et al. [2006] state:

> *Following the practice used in [Spirtes et al., 2000] in our experiments we do not perform an independence test (i.e., we assume independence) unless there are at least five training instances on average per parameter (count) to be estimated.*

I have conducted an empirical evaluation of the impact of choosing either rule, to resolve the Spirtes/Tsamardinos "controversy", and report my findings in this chapter. I found that, typically, it is beneficial to follow the interpretation of Tsamardinos et al. [2006]. Removing edges makes networks sparser, which seems to be approximating "natural" Bayesian networks better than when the keep rule is applied.

6.2 EMPIRICAL EVALUATION

I have examined the impact of the difference in opinion stated by Tsamardinos et al. [2006] and Spirtes et al. [2000] more closely. I have run experiments comparing the running time and accuracy performance of the PC algorithm, implemented in SMILE, where the two implementations under investigation differed only by how they would handle the insufficient data case. I will refer to the advice by Spirtes et al. [2000] as the *keep rule*, and the advice by Tsamardinos et al. [2006] as the *remove rule*.

After running a few initial tests, I found that relatively small datasets showed huge discrepancies between the running time of the algorithm implementing the two rules. I devised a blacklist rule that eliminated the running time difference. I prove the correctness of this rule. Section 6.2.1 presents a detailed explanation of the rule and its proof.

After implementing the blacklist rule, I continued my investigation of the effect of the keep and remove rules, comparing the impact of the rules on the PC algorithm. A detailed description of Experiment 6.1, its methodology, and its results can be found in Section 6.2.2. Finally, I investigated the effect of the keep and remove rules on classification accuracy of Bayesian networks that were learned using the PC algorithm. Section 6.2.3 describes the details of Experiment 6.2.

6.2.1 The Blacklist Rule

In the initial testing phase, I found huge discrepancies between the algorithm running time of the keep rule and the remove rule, when applying them to datasets with a relatively small number of samples. These discrepancies were so large that the required running time prohibited me from running all the tests that I intended to do.

These initial tests involved generating a 10,000 record dataset using a gold standard network, and, applying Equation 6.1 to limit the size S of the dataset in such a way that at a pre-specified conditioning set size C there would be insufficient samples for performing any independence test

$$S = 5 \cdot 2^{C+2} \,. \tag{6.1}$$

This would guarantee the activation of either the keep or remove rule for all cases at that point on. No tests would be able to run and the rest of this phase of the PC algorithm would be completely determined by the applied rule.

My expectations were, when limiting datasets to guarantee rule activation at independence test level 0, i.e., making the datasets so small there would not be enough data to run even a marginal independence test, the keep rule would cause the algorithm to run for a considerable amount of time, while the remove rule would ensure the algorithm would finish swiftly. In either case, no independence test would be performed and any modification of the graph would be due completely to the execution of the keep or remove rule. Respectively, the resulting pattern would be a completely connected, undirected graph, or an empty graph, having all its edges removed.

I found that, indeed, the running time increased when applying the keep rule, but it was much more than I expected. After examining my findings more closely, I found, that when limiting the dataset to the point rule execution is guaranteed at conditioning level 0 (marginal independence tests), the algorithm attempts to try every possible independence test (trying every possible conditioning set) for every edge. Although no test could be run, an exponential number of operations was being performed, causing the long running time of the algorithm.

I have implemented a rule and a heuristic that considerably shortened the running time in this situation:

1. **Blacklist**: If for any edge none of the conditional tests can be performed due to an insufficient sample to cell ratio, this edge is blacklisted and no longer checked in subsequent iterations of the algorithm.
2. **Upper Bound**: The ratio formula can be used to calculate an upper bound on the conditioning set size. This bound is dependent on the minimum variable cardinality and the number of records of the dataset:

$$n \leq \left\lfloor \frac{\log\left(\frac{S}{5}\right)}{\log\left(\min_i Card\left(i\right)\right)} \right\rfloor - 2 \ , \tag{6.2}$$

where S is the sample size and $Card\left(i\right)$ is the cardinality of node i. This bound can be used to terminate the PC algorithm once we can guarantee, using the equation, that

we cannot perform any independence tests without invoking the rule. In the literature I found one case where this heuristic was applied [Stojnic et al., 2012], and expect that others have done so as well.

The upper bound heuristic was not applied in the experiments in this chapter, since once we reach the upper bound, the algorithm terminates, leaving edges behind that otherwise would be subjected to the one of the rules. Although it should not matter when applying the keep rule, combining this heuristic with the remove rule would most likely give results different from applying only the remove rule. I consider it beyond the scope of this chapter to quantify the additional effect of the upper bound heuristic on the quality of patterns produced by the PC algorithm.

I prove that the blacklist rule is correct, i.e., I show that shutting out edges from future independence tests has no negative side effects, since no test of independence would be able to be performed on them anyway:

Let $I(X, Y|\mathbf{Z})$ be an independence test of an edge between variables X and Y. This test is conditioned on variable set $\mathbf{Z}$, where $|\mathbf{Z}| = n$, and $\mathbf{Z}$ contains only adjacent variables of X, $\mathbf{Z} \subseteq \mathbf{Adj}(X)$ and $Y \notin \mathbf{Z}$. Every independence test $I(X, Y|\mathbf{Z})$, requires a contingency table T with N elements. N is the product of the variable cardinalities:

$$N = Card(X) \cdot Card(Y) \prod_{i=1}^{n} Card(Z_i) \tag{6.3}$$

The ratio R_n is defined as:

$$R_n = \frac{|\mathbf{D}|}{N} = \frac{|\mathbf{D}|}{Card(X) \cdot Card(Y) \prod_{i=1}^{n} Card(Z_i)} = r \ , \tag{6.4}$$

where $\mathbf{D}$ is the dataset, and r is the predefined minimum ratio to successfully perform an independence test.

Theorem 6.2.1. *If for an edge between X and Y every independence test $I(X, Y|\boldsymbol{Z})$ fails to satisfy ratio r: $R_n < r$, for every possible conditioning set $\boldsymbol{Z}$ of size n, it will also fail to satisfy ratio r for any test with any conditioning set $\boldsymbol{Z}$ of size $n+1$: $R_{n+1} < r$.*

Proof. If $|\mathbf{Z}| = n + 1$, then:

$$\begin{aligned} R_{n+1} = & \frac{|\mathbf{D}|}{Card\,(X) \cdot Card\,(Y) \prod_{i=1}^{n+1} Card\,(Z_i)} \\ = & \frac{|\mathbf{D}|}{Card\,(X) \cdot Card\,(Y) \cdot Card\,(Z_{n+1}) \prod_{i=1}^{n} Card\,(Z_i)} \\ = & \frac{R_(n)}{Card\,(Z_{n+1})} \end{aligned}$$

If $R_n < r$, then clearly $R_{n+1} = \frac{R_n}{Card(Z_{n+1})} < r$ since $Card\,(Z_{n+1}) \geq 1$ □

6.2.2 Experiment 6.1: Comparing the Performance of the Keep Rule versus the Remove Rule

Having justified the use of the blacklist rule for the keep rule, I continued the experiment where I compared the performance of the keep rule with the remove rule. My expectation was that due to the sparse nature of most networks, the remove rule would allow for better performance than the keep rule, and the results of the experiment confirmed these expectations.

6.2.2.1 Methodology Similar to Experiment 5.1, I performed a gold standard task with nine different networks, listed in Table 6.1. Experiment 6.1 consisted of performing the following steps 100 times and collecting all results:

1. Randomly generate a 10,000 record sample from the gold standard network.
2. For each of the conditioning size limits C in $[0, 9]$:
   a. Reduce the sample size of the data to S, where S is calculated by inputting C in Equation 6.1.
   b. Learn a model from the data using the PC algorithm using the keep rule or the remove rule.
   c. Calculate the Hamming distance from the learned models to the gold standard.
   d. Calculate the Hamming distance between the learned models.
   e. Calculate the Skeletal distance from the learned models to the gold standard.
   f. Calculate the Skeletal distance between the learned models.

Table 6.1: Gold Standard Evaluation Networks used for Experiment 6.1

Network	Nodes	Edges	Reference
Asia	8	8	Lauritzen and Spiegelhalter [1988]
Nursery[2]	9	10	Ratnapinda and Druzdzel [2013]
Adult[2]	15	18	Ratnapinda and Druzdzel [2013]
Bank[2]	17	28	Ratnapinda and Druzdzel [2013]
Chess[2]	37	100	Ratnapinda and Druzdzel [2013]
Alarm	37	46	Beinlich et al. [1989]
Hailfinder	56	66	Abramson et al. [1996]
Hepar	70	123	Onisko [2003]
CPCS179	179	239	Pradhan et al. [1994]

g. Count the number of edge orientation mistakes between the learned models and the gold standard.

h. Count the number of edge orientation mistakes between the learned models.

i. Store the number of times the rule were applied in both cases.

j. Store the running time of the algorithm in both cases.

6.2.2.2 Results The results from Experiment 6.1 confirmed my expectations that, due to Bayesian networks typically being sparse, the remove rule outperformed the keep rule, albeit mostly with a marginal difference. Figure 6.1 shows the gold standard recovery performance of the two rules. Additionally, the Hamming distance between the two constructed patterns is shown in the graph as well.

The difference between the two rules is most apparent when there is very little data available. In the worst case, the keep rule results in a completely connected graph, and the remove rule results in an empty graph without edges. The remove rule capitalizes on the

[2]These networks were created using data from the UCI Machine Learning Repository [Bache and Lichman, 2013]

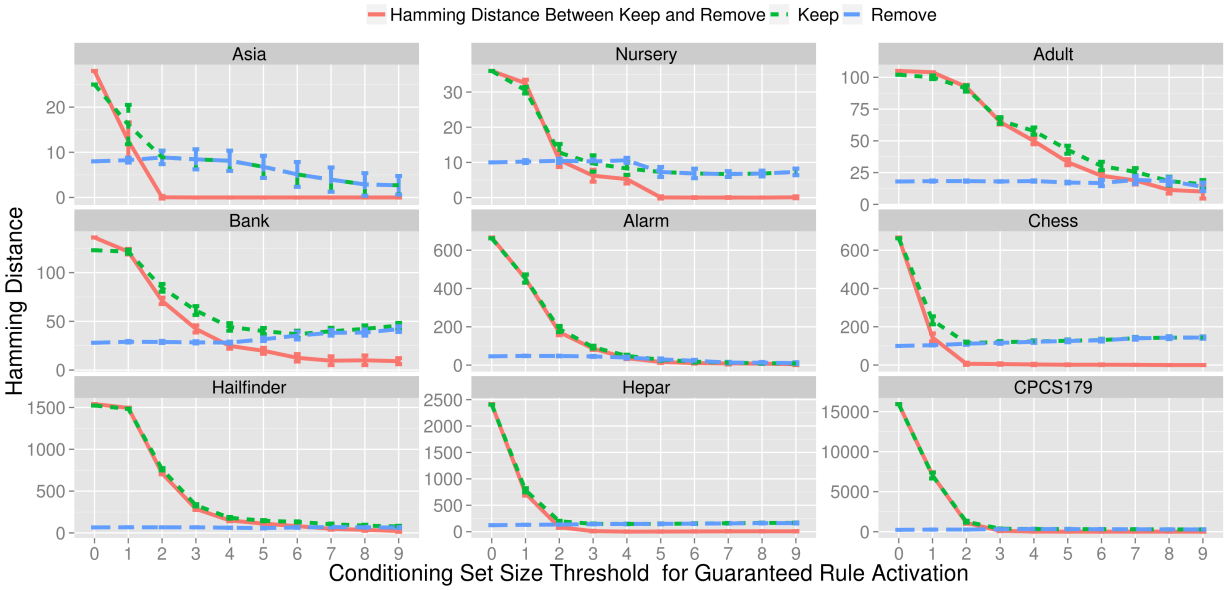


Figure 6.1: Gold Standard Recovery Results of Experiment 6.1

natural sparsity of Bayesian networks. Thus, its worst case scenario is much closer to the gold standard network than the worst case scenario of the keep rule.

Once more data is available, the algorithm will be able to perform more independence tests successfully and the difference between the two results quickly disappears. The remove rule outperforms or performs equally to the keep rule variant. The Hamming distance is consistently lower or equal. Figure 6.2 shows that in most cases the skeletons of the results tend to converge when we are able to perform independence tests with conditioning set sizes of 2 or more, the remaining discrepancy between result patterns of the rules is due to edge orientation differences.

Figure 6.3 illustrates the edge orientation mistakes that were made by both algorithm variants and shows how many edges differed between the patterns. Overall the remove rule makes fewer mistakes, but it is important to note that the measure only counts cases where in both patterns (result and gold standard, or both results) an edge exists, which is why in the first few conditioning limits, the number is quite low, starting with 0 in the first step. In that case, the remove rule results in an empty pattern, meaning there are no incorrect

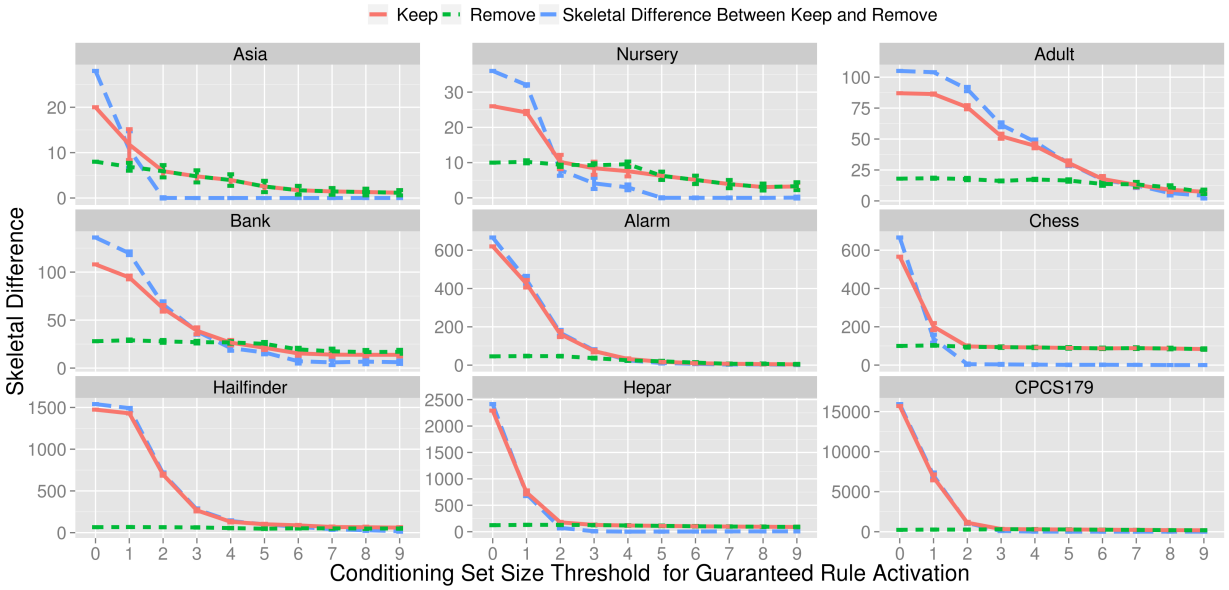


Figure 6.2: Skeletal Differences Between Patterns and Gold Standard

edges. However, when there is more data available, the algorithm still tends to make less orientation mistakes when the remove rule is applied.

The analysis of the algorithm running time showed that there was no difference between applying either of the two rules. The results are illustrated in Figure 6.4. In contrast, I did observe a difference in the number of times a rule was applied in each case. Figure 6.5, having a logarithmic y-axis, shows that when the keep rule is used, a much larger number of independence tests requires application of the keep rule due to insufficient data.

This difference in number of rule applications is easily explained. Applying the remove rule achieves two things: 1) the removal of the edge means it can never be tested again, 2) The two nodes that were previously connected by this edge can no longer appear as conditioning variables in each of their remaining conditioning independence tests.

Removing edges thus decreases the number of edges that need to be tested in subsequent iterations of the algorithm, limits the number of possible future conditioning sets, and reduces the possibility of not having sufficient data to run an independence test.

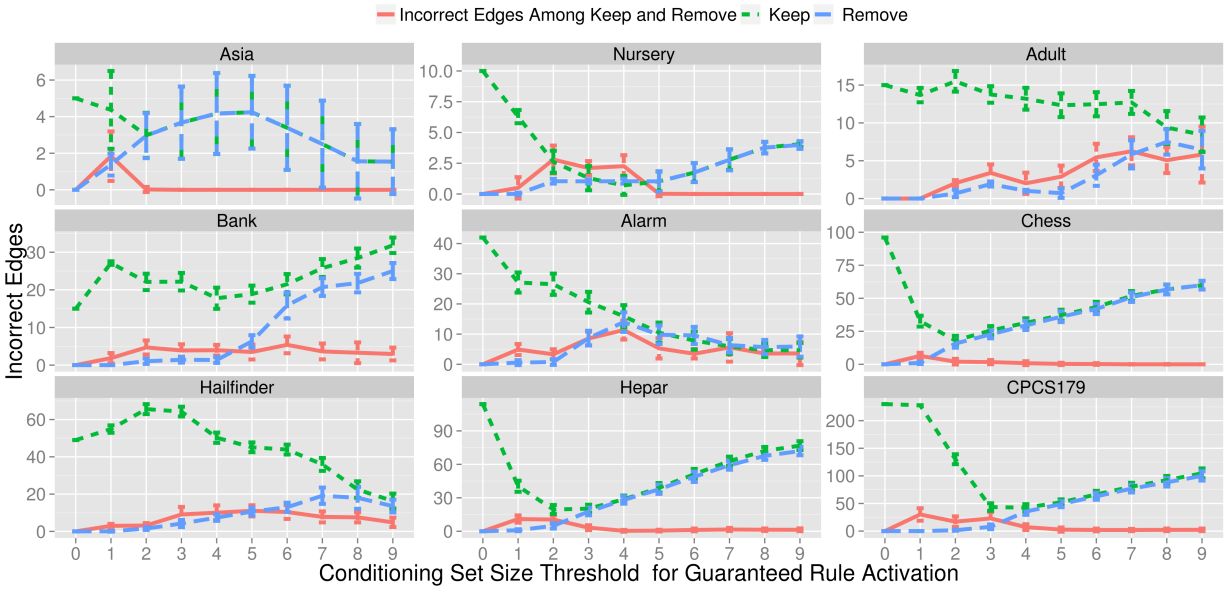


Figure 6.3: Incorrect Edges Between Patterns and Gold Standard

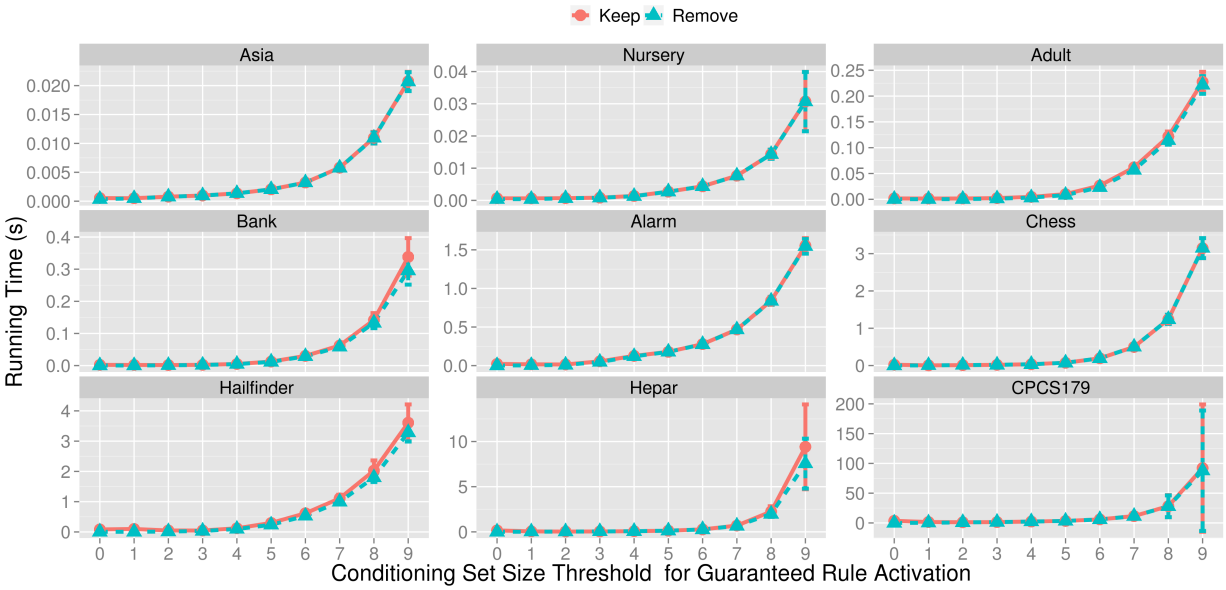


Figure 6.4: Average Running Times for Experiment 6.1

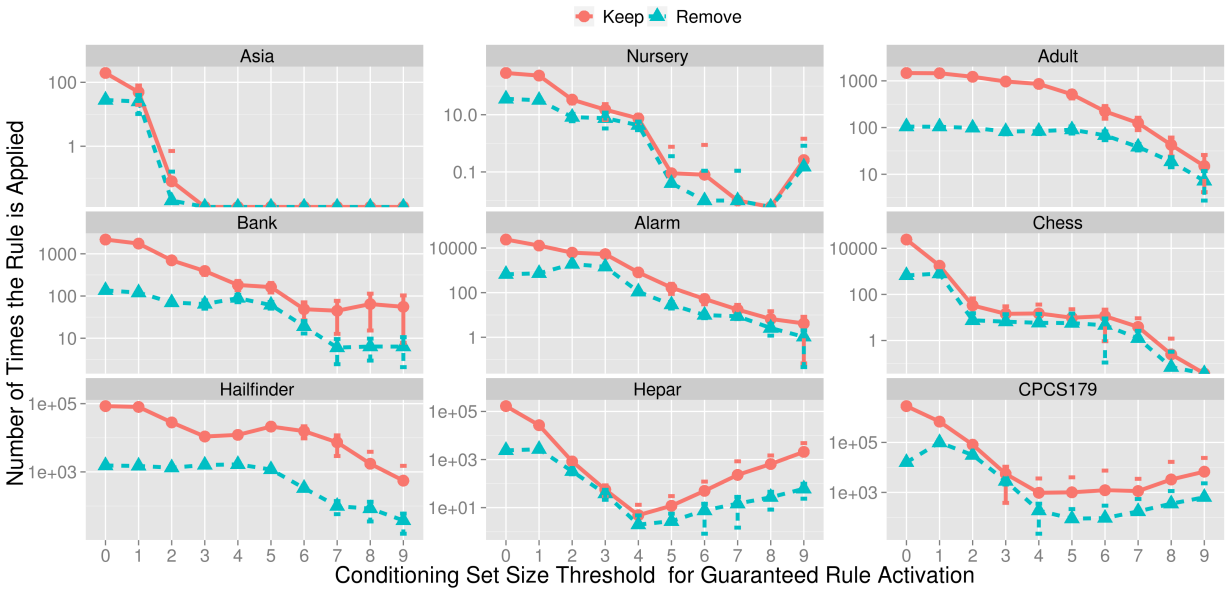


Figure 6.5: Average Number of Rule Applications for Experiment 6.1

To summarize, the remove rule exploits the natural sparsity of Bayesian networks, and performs better or equal to the keep rule when recovering the structure of a gold standard network, especially when fewer data is available. However, removing edges does not necessarily improve the running time of the algorithm more than applying the keep rule. The blacklist rule, used in conjunction with the keep rule, eliminated all running time discrepancies.

In Experiment 6.2, we will consider the performance of the keep and remove rule when, instead of a gold standard network recovery task, we consider the typical machine learning task of classification.

6.2.3 Experiment 6.2: Rule Performance when Executing a Classification Task

When comparing the performance of the two rules on a gold standard recovery task, I found the remove rule had an edge over the keep rule, especially when there was less data available. But, the gold standard recovery task limits our evaluation to just the structure of the BN, omitting the other main component, the (conditional) probability distribution tables needed

for each node to fully define a Bayesian network. To provide a more complete picture, I have performed an experiment to test the impact of the rules on the classification accuracy of a model learned with the PC algorithm.

6.2.3.1 Methodology For the classification task I chose twelve example datasets from the UCI machine learning repository [Bache and Lichman, 2013], the datasets are listed in Table 6.2. Experiment 6.2, consisted of performing the following steps:

Table 6.2: Classification Datasets used for Experiment 6.2

Datasets	Variables	Records	Classes
Hayes-Roth	4	132	3
Car	6	1,728	4
Nursery	9	12,960	5
Tic-Tac-Toe	10	958	2
Zoo	17	101	7
Bank	17	45,211	2
Breast-Cancer	22	80	2
SPECT	23	80	2
Soybean	36	266	15
Chess	37	3,196	2
Connect-4	43	67,557	3
Semeion	257	1,593	10

1. Randomize the order of the records in the dataset.
2. Run a 10-fold cross validation with the following steps:
   a. Divide the dataset into a training and a test set.
   b. Based on the same principle as in Experiment 6.1, reduce the training dataset to control for the activation frequency of the keep and remove rule, setting the conditioning

limit parameter C to $[1, 3]$.

c. Learn patterns from the training data using the PC algorithm with the keep and the remove rule.
d. Transform patterns into DAGs using the conversion method implemented in SMILE.
e. Learn parameters of the networks using the training dataset and the EM algorithm.
f. Determine the classification accuracy of each network by predicting the class variable using samples from the the test dataset, and verify the predictions with the actual class values.
g. Store the number of times the rules were applied in both cases.
h. Store the running time of the algorithm in both cases.
i. Store the time required to run inference on the Bayesian network in both cases.
j. Store the total clique size of the junction tree used for inference.

The dataset sizes are limited to guarantee that a large number of independence tests will be unable to be performed due the insufficient data. This in turn guarantees the two rules will be applied frequently. Rule application drops to near zero when there are sufficient data available in most cases, which is why for Experiment 6.2 data was limited to allow at most two conditioning variables.

6.2.3.2 Results One third of the datasets used in Experiment 6.2 failed to produce a tractable Bayesian network. Although the PC algorithm produced a pattern applying either rule, and the patterns were successfully converted into DAGs, a common problem was that nodes would have very large parent sets. This, in combination with nodes having a considerable number of variable states, resulted in the conditional probability tables (CPTs) becoming too large, causing the experiment to stall in the conversion process. This only occurred when the PC algorithm applied the keep rule when learning the pattern, and typically when the data was constrained the most by the limitation process that regulates rule activation for the two experiments. Nine of the twelve datasets finished their run without problems. For these datasets the results are shown in Figures 6.6-6.9.

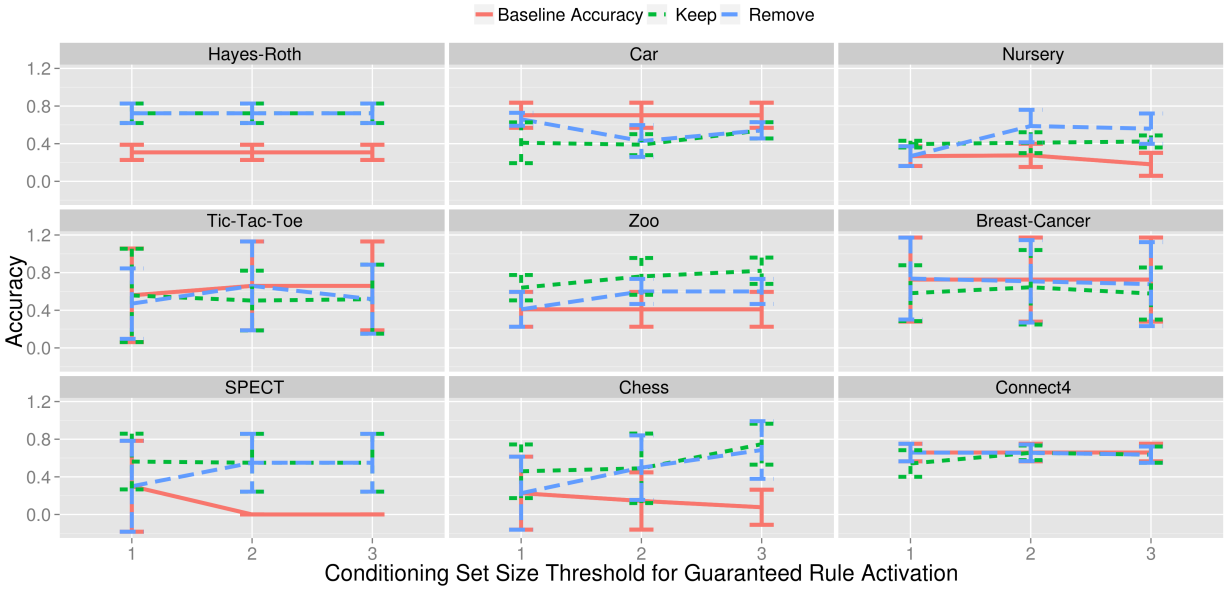


Figure 6.6: Average Classification Accuracy Results for Experiment 6.2

Table 6.3: Wilcoxon Signed Rank Test Results for Classification Accuracies of Experiment 6.2

	Remove	Baseline
Keep	0.8563	**0.03345**
Remove		**0.03694**

Following the procedure described by Demšar [2006], I performed Wilcoxon signed rank tests [Wilcoxon, 1945] after ranking the performance of the algorithms to determine if either of the rules significantly outperformed the other. Additionally, I tested if the rules outperformed the baseline accuracy model, which simply learns the class label proportion from the training data and applies the label with the maximum probability to all samples. For the Wilcoxon test I considered 27 paired data points (9 datasets, 3 constrained versions). These points were the ranks of the algorithms after comparing their classification accuracy, averaged over the 10 folds. I found that, with respect to accuracy, there were no significant

differences between the performance of the algorithms applying either of the rules. The performance of the algorithm did differ statistically significant from the baseline performance, but the evidence is weak.

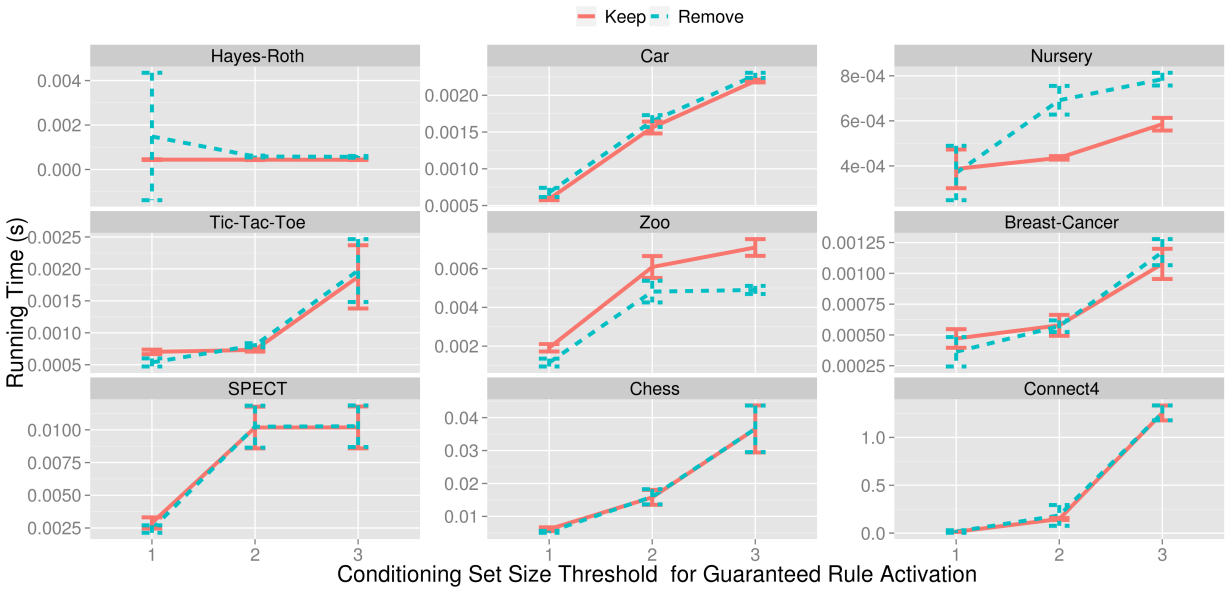


Figure 6.7: Average Running Times for Experiment 6.2

Figure 6.7 shows the running time needed by the algorithms to learn the models. For the datasets tested (specifically, their three restricted versions), neither of the rules drastically influenced the running time, which is supported by the result of the Wilcoxon test I performed on the algorithm running times. No significant difference was found between the algorithms ($V = 139$, $p = 0.2386$). The inference time, shown in Figure 6.8, defined as the average time it takes for the network to perform inference on a sample (with SMILE's inference implementation), does show a significant difference. The one-sided Wilcoxon test indicated the remove rule outperformed the keep rule ($V = 349.5$, $p = 5.247\text{e-}06$). But, from a practical point of view, the performance differences are only noticeable when the models were learned with very little data.

The networks that were learned with the very limited datasets were much denser, and as a result the nodes would typically end up with larger CPTs. When performing inference,

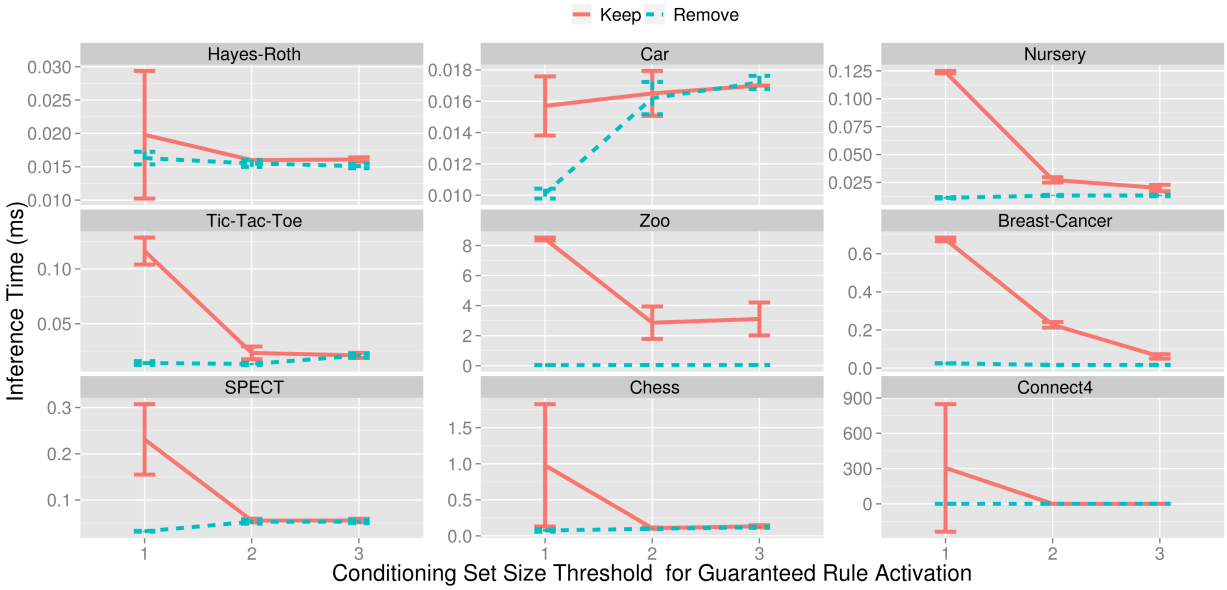


Figure 6.8: Average Inference Times for Experiment 6.2

SMILE converts the networks into junction trees. As was discussed in Section 2.2.1.2, cliques are created from the moralized and triangulated Bayesian network. Nodes are always put in the same clique as their parents. Denser networks, where nodes have more parents thus result in larger cliques. The total clique size, i.e, the sum of all elements of all the clique potentials, gives an indication of how much time inference will take. Figure 6.9 shows the total clique sizes for the networks. Larger clique nodes, that have to be created for the junction tree, take more space and require more time for construction and inference calculations. With classification accuracy performance not differing much between the two rules, in general, it seems wise to apply the remove rule for handling cases with insufficient data. Networks will be sparser, require less parameters, perform inference faster, and provide similar classification accuracy.

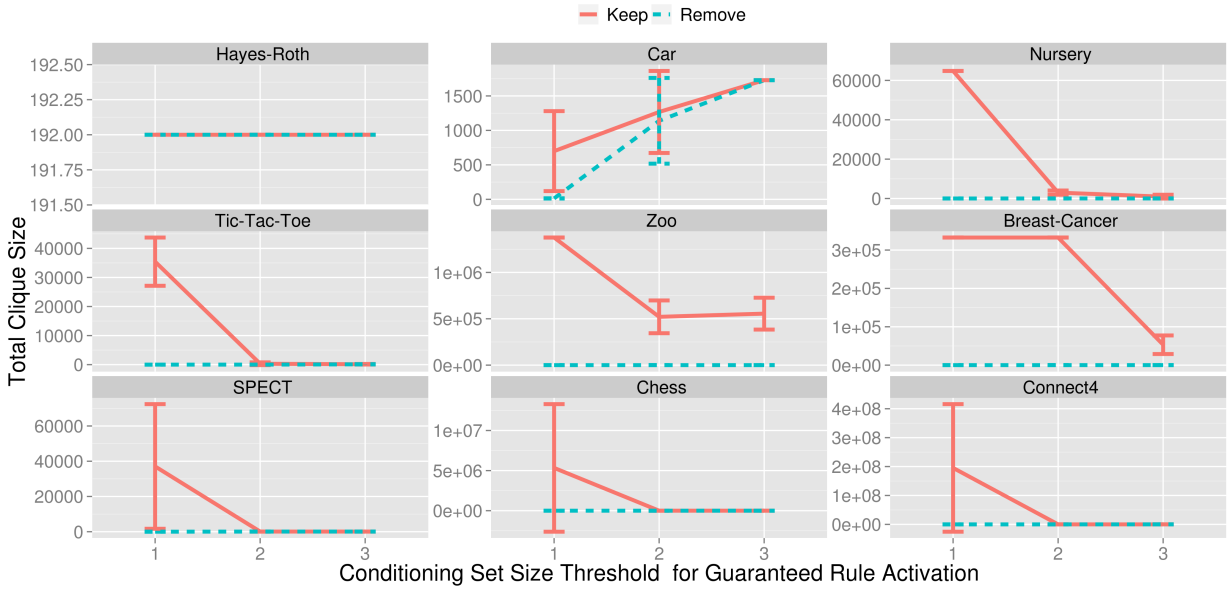


Figure 6.9: Average Total Size of All Cliques in the Network for Experiment 6.2

6.3 DISCUSSION

The difference in the advice given by Spirtes et al. [2000] and Tsamardinos et al. [2006] sparked the question: "does picking either of the rules impact the performance of the algorithm?" I investigated this question by running two types of experiments, 1) Recovering gold standard networks and 2) learning classification models and assess their accuracy.

To control for the effect of different dataset sizes on the number of times the rules would need to be applied, I made use of a simple formula to determine the proper dataset size to induce the insufficient data problem reliably at predefined conditioning set sizes.

One immediate consequence of controlling the size of the datasets used in Experiments 6.1 and 6.2, was that I found that with very little data, the keep rule, as described by Spirtes et al. [2000] would cause the algorithm to run for an unacceptable amount of time, even when the number of variables in the dataset was still small.

As a result of this observation, I developed a rule that would blacklist edges after all possible independence tests in a round of tests failed to satisfy the preset ratio requirements

due to insufficient data. This completely solved the running time issue. I proved that the rule has no negative impact on algorithm performance.

One of the main takeaways from the results of Experiment 6.1 is, that with sufficient data the difference between the algorithm results due to the two rule options, disappear almost completely. The difference in Hamming distance decreases fast. In the cases where we limit the data to a very small amount, the remove rule has a better score than the keep rule. This is due to Bayesian networks commonly being very sparse. When applying the keep rule we end up with much denser networks and these will typically differ more from the original gold standard network.

If we assume that Bayesian networks should be sparse, which has certain benefits, then choosing to let the PC algorithm apply the remove rule, should be the optimal course of action. When we are confronted with datasets with less data, where we would most likely need to apply the rule many times, we should end up with more practical and potentially better networks this way. And, if there is sufficient data available we can be reassured that choosing the remove rule, with the benefits of sparsity that it provides, will let the PC algorithm produce a Bayesian network structure that would have been very similar to one produced by using the keep rule.

Additionally, the rules do not effect the running time of the PC algorithm differently. Controlling for the number of variables, the running times when applying either rule for solving insufficient data cases, seems to be determined completely by the size of the dataset.

In Experiment 6.2, I applied the PC algorithm and the two rules to the problem of learning a model from training data, and test the algorithm performance using a test dataset. In addition to running the PC algorithm to learn a pattern, which represents an equivalence class of Bayesian network structures, I ran a conversion algorithm present in SMILE (based on the algorithm by Dor and Tarsi [1992]) to convert the pattern into a DAG. After this step, parameters for the CPTs of the nodes are learned, and at that point we can perform the classification task on the test data.

I found that a common problem is that the PC algorithm and the DAG conversion algorithm together produce DAGs where nodes have too many parents. This results in the CPTs for the nodes to become unacceptably large, causing the experiment code to stall,

taking up all of the computer's memory, and forcing the operating system to swap data from the memory to disk. At this point, the process running the algorithm was canceled. From the twelve datasets selected for Experiment 6.2, nine ran without problems, the other three had this problem. Commonly, running the PC algorithm on the datasets would result in networks with large parent sets, with examples where nodes would have all other nodes as parents, a problem exacerbated further by variables having many states. What we can observe from this problem, is that, especially when datasets have few samples, the PC algorithm frequently returns patterns which may not end up being feasible Bayesian networks. In Chapter 7, I discuss several approaches to modifying the result of the PC algorithm to produce Bayesian networks that will be more tractable in model size and when performing inference on the networks.

The results from Experiment 6.2 indicated that neither of the rules outperforms the other with regards to running time and classification accuracy. Chapter 7 discusses structure modification approaches that aim to enhance classification performance.

Significant differences were found between the rules when comparing the algorithm performance on inference running time, and total clique size (when converting the network into a junction tree). Applying the remove rule guarantees sparser networks, which in turn will be beneficial when running inference on the networks. The clique nodes in the junction tree require less parameters, which should translate into the inference algorithm requiring less time to classify samples.

Summarizing, my results indicate that the remove rule should be preferred over the keep rule. With enough data, the effects of the rules are minimal. But, with smaller datasets, the rules effect the algorithm result more significantly. The remove rule will result in sparser networks than when the keep rule is applied. Bayesian networks are typically sparse, and applying the remove rule would result in more "natural" networks. The added benefit of sparse networks is that they are usually more tractable. Sparser networks require less parameters, making inference more efficient by requiring less memory and less time. The rules did not significantly influence the classification accuracy of the PC algorithm. No significant differences were found between the rules and between either rule and the baseline model. The observed problems with patterns learned from the PC algorithm not being

converted to tractable Bayesian networks, allows for opportunities to investigate optimal modifications to the algorithm and the general procedure to guarantee tractable networks. This is the topic of the next chapter.

7.0 IMPROVING TRACTABILITY OF BAYESIAN NETWORKS DERIVED FROM PATTERNS

This chapter proposes several approaches to modifying patterns outputted by the PC algorithm or DAGs converted from patterns, improving their tractability and performance as classifiers.

7.1 INTRODUCTION

In the previous two chapters, we have come across situations in their respective empirical evaluations, where we were not able to perform classification experiments on some datasets due to Bayesian networks being intractable after converting them from the patterns obtained from the PC algorithm. What these intractable networks all had in common, was that they all had nodes with large parent sets. Nodes with such sets require large CPTs, as these grow exponentially with the number of parents. With 30 or more parents, nodes require billions of CPT entries, resulting in large memory usage for model storage and inference. Such networks typically do not fit in the working memory of a common computer, causing the problems observed in my experiments.

The memory problems occurred mostly in networks learned using the keep rule. Due to the nature of the keep rule, not removing edges when confronted with insufficient data for an independence test, networks will be more dense. Especially when few data records were available, many parent sets of these networks were large, sometimes containing all variables. Figures 7.1 and 7.2 show for several datasets the types and quantity of edges the nodes in the patterns contain, and the effect on the proportions of these edges after converting the patterns

into DAGs. Patterns constructed by the PC algorithm contain three different types of edges, 1) directed, 2) undirected, and 3) bi-directed edges. In a pattern, a node can be involved into four different relationships. A node can be a parent, a child, be adjacent to another node, or be linked to a node by a bi-directed edge, which signifies a hidden common cause influencing both nodes connected by the bi-directed edge, an artifact of the PC algorithm. DAGs only have directed edges, nodes can only be parents or children. When we convert a

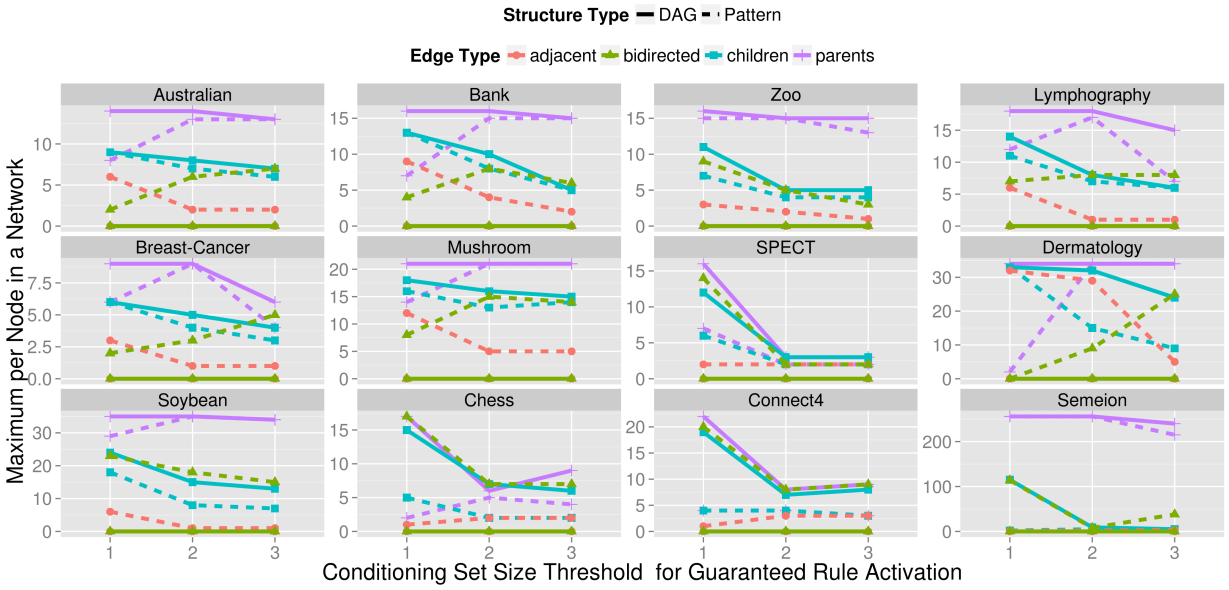


Figure 7.1: Maximum Occurrence of Edge Type in a Pattern or a DAG, Keep Rule

pattern into a DAG, we must decide on what to do with the undirected and bi-directed edges. Converting the undirected edges is relatively simple. A pattern represents an equivalence class of Bayesian networks, and, when satisfying certain constraints, the undirected edges can be replaced by directed edges that are oriented in either way. Dor and Tarsi [1992] proposed one example of an algorithm that can convert a pattern into a valid DAG. The SMILE library applies this algorithm to convert patterns into DAGs. Additionally SMILE converts bi-directed edges into directed edges based on a partial node ordering, derived from the result produced by the [Dor and Tarsi, 1992] algorithm, when it is presented the pattern without bi-directed edges.

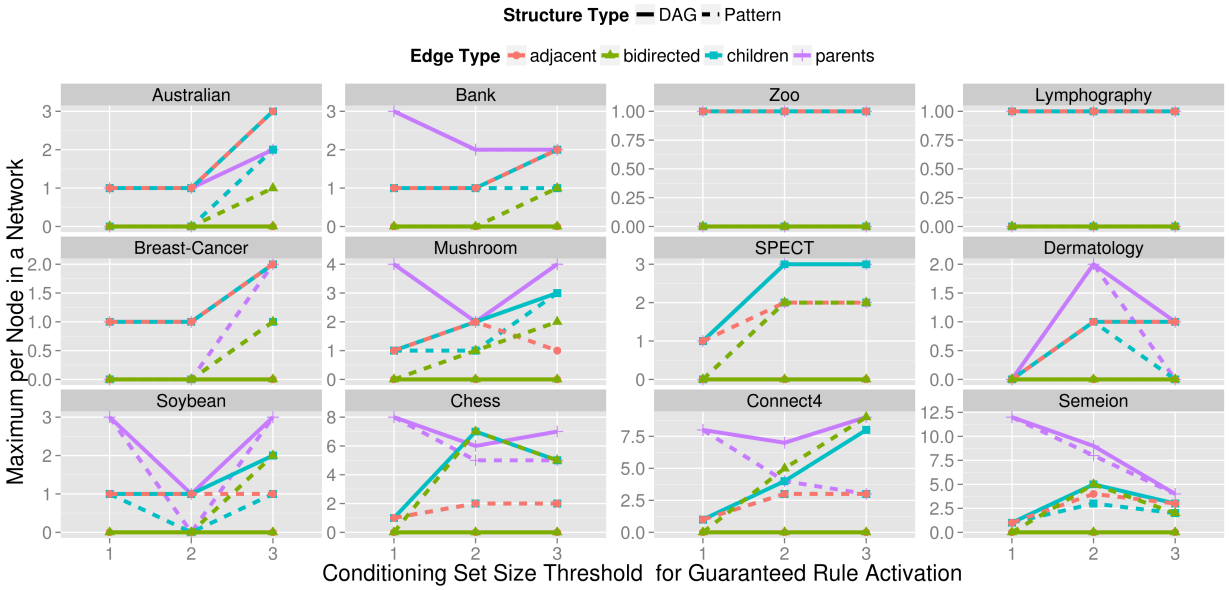


Figure 7.2: Maximum Occurrence of Edge Type in a Pattern or a DAG, Remove Rule

Figures 7.1 and 7.2 show the per node maxima of each edge type when applying the keep and remove rules. The difference between the two, when taking the eventual impact of the network structure on the size of the CPTs, is enormous. The maxima are shown to make the point that it takes only one extreme case to make the whole network intractable for inference purposes such as a classification task.

From the figures, we see that bi-directed edges tend to occur more frequently than undirected edges and make a potentially larger contribution to the parent set. In Section 7.3, I propose to remove all bi-directed edges from the pattern and investigate empirically the effect of this suggestion on the gold standard recovery task and classification tasks. The results of this evaluation show that removing bi-directed edges does not seem to have a significant (negative) effect on the performance of these two tasks (although any change that promotes network sparsity tends to improve the gold standard recovery task result), but it does not provide a definite solution for network intractability.

It is still possible for a node in a pattern to have a large parent set, which will result in an intractable Bayesian network. To guarantee network tractability, we must make more

significant modifications to the pattern to curb the CPT sizes of the Bayesian network. We can accomplish this by limiting the number of parents that a node is allowed to have. This action, similarly to removing bi-directed edges, permanently alters the pattern or DAG structure. Limiting the number of parents is accomplished by selecting a subset of nodes from the complete parent set. If we limit the number of parents to k and if a node has a parent set of size n, then there are $\binom{n}{k}$ possible subsets we can choose. Faced with this choice, we would like to make an optimal decision. In Section 7.4, I describe four different heuristics to picking an optimal restricted parent set. I report the results from an empirical evaluation, where I compare the performance of the four heuristics on a gold standard recovery and a classification task, and report the impact of three different parent set sizes on task performance. I found that applying these approaches improved the tractability of the network, decreased the Hamming distance in the gold standard recovery task, and, in general, does not negatively affect the classification accuracy of networks in the classification task.

Finally, in Section 7.5, I discuss three approaches that aim to improve the classification accuracy of a network. These methods apply knowledge from typical BN classification approaches to modify the raw BN structure outputted by the PC algorithm. I evaluate the performance of these methods against the original network, the baseline accuracy, and the performance of a naive Bayes classifier as an additional baseline. The results from the evaluation suggest that it is possible to improve the classification accuracy of a model by making structural changes, but the extent of the changes that are necessary to obtain this increase in accuracy diminishes the usefulness of the general idea of this approach.

7.2 RELATED WORK

In this section, I discuss relevant literature on limiting parent sets and the Bayesian networks used for classification.

7.2.1 Limiting the Number of Parents of a node

Quite commonly, the Bayesian network learning literature, when discussing structure learning, points out the tractability problem caused by nodes with large parent sets [Silander and Myllymaki, 2012, Friedman and Koller, 2000, De Campos et al., 2009, Neil et al., 1999, for instance]. But, typically, it seems that after the problem has been identified, the method of parent limitation is left as an exercise to the reader. Two examples of approaches to constraining the parent set are presented below.

In their exact Bayesian network learning algorithm, Silander and Myllymaki [2012] apply a Bayesian method to determine the best parent set. Their method recursively grows the best set of parents by calculating the Bayesian score for the current parents set $\mathbf{S}$ and the current candidate. Then, after scores are calculated for each candidate, the best one is added to $\mathbf{S}$.

De Campos et al. [2009] propose an exact Bayesian network structure learning algorithm that allows for the definition of parameter and structural constraints. The parameter constraints are based on the work by Wellman [1990]. Structural constraints allow the user to forbid the existence of edges, and to limit the indegree of a node, i.e., the size of its parent set. Exact structure learning algorithms, in the worst case, search the complete space of Bayesian network structures. Typically, optimization techniques such as *branch-and-bound* are used to eliminate suboptimal network structures from the search. In the algorithm proposed by De Campos et al. [2009], the branch-and-bound technique is used and the constraints, to be specified by the user, are enforced during the sufficient statistic calculation phase. The algorithm pre-calculates a large portion of the necessary sufficient statistics. The statistics are checked against the constraints and only those that satisfy all constraints are stored in the cache used for calculating structure scores. This eliminates networks from the search

that, for instance, would have violated the indegree constraints. If no statistics for oversized parent sets are stored in the cache, they cannot be considered in the search as valid BN structures.

7.2.2 Bayesian Networks Classifiers

Bayesian networks have been used extensively for the classification problem. The best known example is the naive Bayes classifier, which can be represented by a Bayesian network as a class node and a set of feature nodes, where the class node is the parent of all feature nodes. The naive Bayes classifier makes very strong independence assumptions, but typically performs quite well. Work has been done on BN classifiers with more complex structures or more complex inference [Friedman and Goldszmidt, 1996, Friedman et al., 1997, Dash and Cooper, 2002]. Two enhancements of the naive Bayes classifier are the Tree-Augmented Naive Bayes classifier (TAN) [Friedman et al., 1997] and the Augmented Bayesian Network classifier (ABN) [Friedman et al., 1997].

The difference between discriminative and generative learning has been discussed extensively [Friedman et al., 1997, Roos et al., 2005]. Generative learning is the "standard" approach to learning where typically a joint likelihood is optimized for parameter and/or structure learning. Discriminative learning is more specialized to classification, here we optimize a conditional likelihood instead.

Constrained Bayesian network learning algorithms such as the one proposed by De Campos et al. [2009] can be applied to learn NB, TAN, or ABN classifiers by specifying the appropriate structural constraints.

The work in this chapter describes approaches that modify a Bayesian network structure or pattern in the final phase, i.e., after a network has been learned from data, we modify the structure to enhance its performance. Lam [1998] discusses a similar problem where an existing Bayesian network is refined with new data, using a Minimum Description Length (MDL) approach. The approaches in this chapter are not full-fledged algorithms, but heuristics that modify the network structure with or without the use of the dataset that was used for learning the structure.

7.3 REMOVING BI-DIRECTED EDGES

The first heuristic, aims to improve the tractability of Bayesian networks constructed by the PC algorithm by removing all bi-directed edges from the pattern before converting the pattern to a DAG. Bi-directed edges are an artifact of the PC algorithm and can be interpreted as the case where the algorithm cannot determine the direction of the arcs between the two nodes. It differs from the undirected edge, also present in patterns, by suggesting the possibility that the algorithm could not decide the direction of the arc due to it not being a direct dependence, but an indirect dependence through a hidden common cause [Spirtes et al., 1993].

I have experimented with removing the bi-directed edges from patterns in a gold standard recovery task and a classification task. I found that removing the bi-directed edges positively affected the gold standard recovery task (mainly due to the increased sparseness of the modified network), and that the classification task did not show changes in classification accuracy, but the increased sparseness of the graph contributed to a decrease in the required inference time and the total clique size of the junction tree used for inference.

7.3.1 Experiment 7.1: Gold Standard Recovery

The methodology used for the gold standard recovery task is identical to Experiment 6.1, described in Section 6.2.2. Figure 7.3 shows the main results of Experiment 7.1: the Hamming distances for the different dataset limitations, comparing the performance of the PC algorithm, applying the keep rule, with and without the extra step that removes all bi-directed edges from the pattern.

I found that the Hamming distances of the patterns with or without bi-directed edges differ significantly. The Pattern without bi-directed edges is consistently closer to the gold standard pattern (converted from the original network) than the original pattern (one-tailed Wilcoxon signed rank test, $V = 3081$, $p = 8.575\text{e-}15$). This result is easy to explain. The keep rule has the tendency to produce denser networks by making conservative decisions when faced with insufficient data. The PC algorithm then might not be able to determine clear

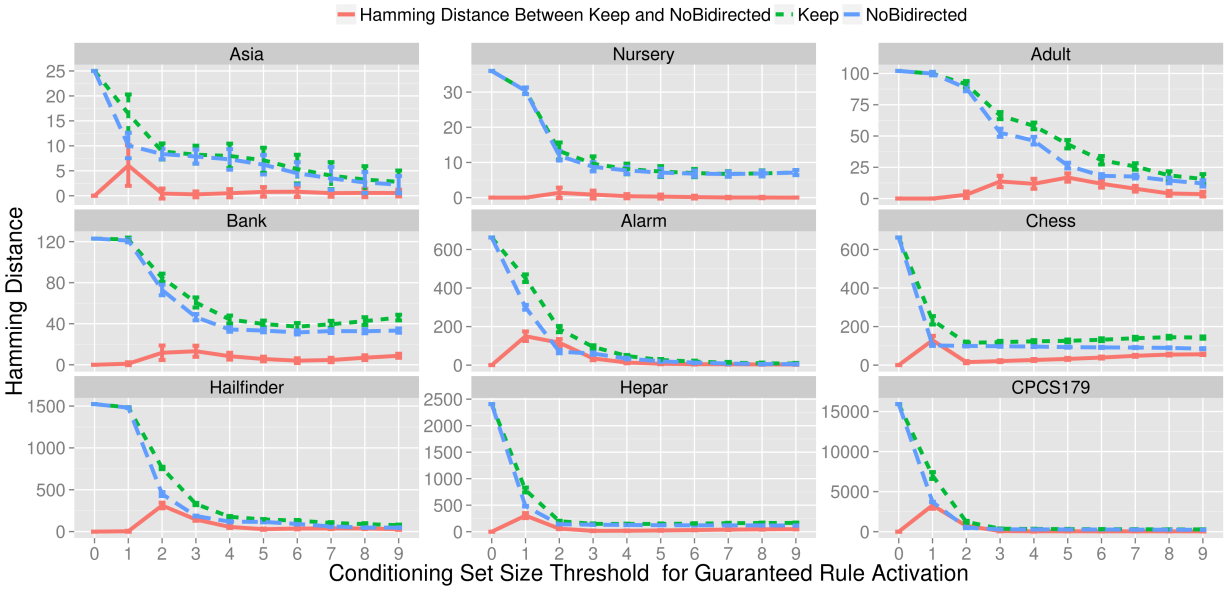


Figure 7.3: Gold Standard Recovery Results for Experiment 7.1

orientation for these extra edges, potentially turning them into bi-directed edges. Removing these edges then reduces "waste" and we may end up with a better pattern.

Another, explanation is that by removing edges we make the pattern more sparse, eliminating opportunities for edge orientation mistakes and spurious edges. As long as we eliminate more edge orientation errors and extra edge errors than erroneously remove actual edges, we end up with a better result in the end. In cases with little data, we might still be willing to make these assumptions to get a better guess for a causal structure, and a potentially more tractable network.

7.3.2 Experiment 7.2: Classification Accuracy

The second part of the evaluation of the "remove bi-directed edges" heuristic, was a classification task. The methodology used for this task is identical to Experiment 6.2, described in Section 6.2.3. Figure 7.4 shows the result of running a 10-fold cross-validation on the nine datasets. Similar to the results of Experiment 6.2, discussed in Section 6.2.3, I found that there are no significant differences between the classification accuracy for the classifiers

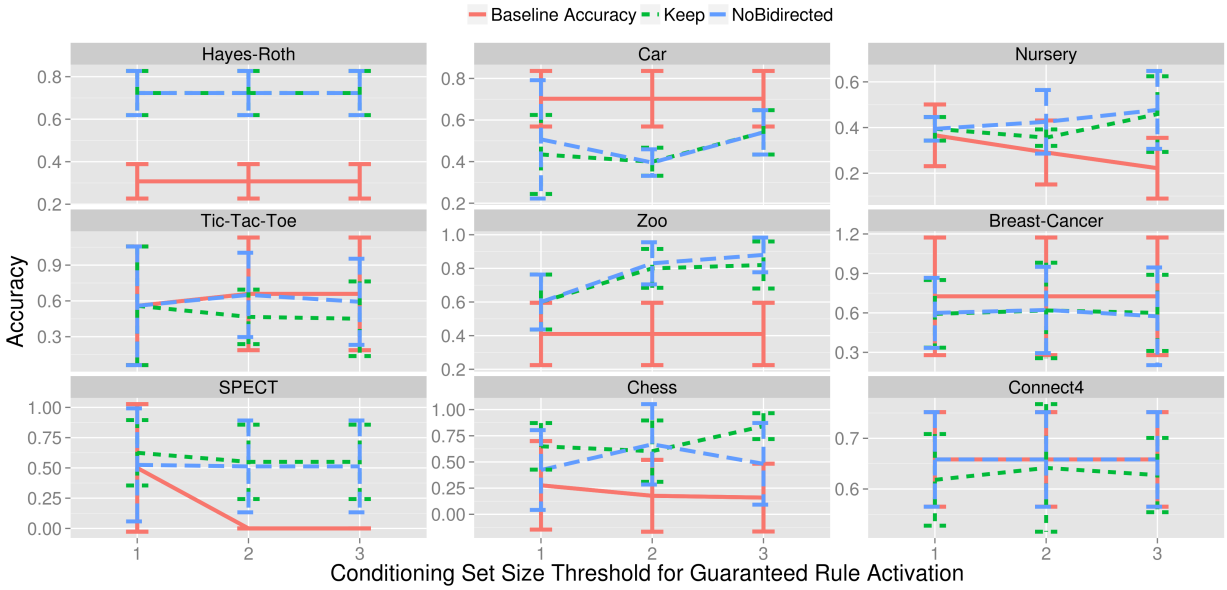


Figure 7.4: Average Classification Accuracy Results for Experiment 7.2

learned using the PC algorithm with or without removing bi-directed edges from the result pattern before translating it into a DAG. Statistically, there is no difference between the classification performance of the original and the modified structure (V = 82, p = 0.4009). For both the inference time, and the total clique size for the junction tree used for inference, we do find differences between the two. Since these results and their cause are effectively the same as the results in Section 6.2.3, they are not illustrated here, but the inference time (V = 368, p-value = 8.989e-06) and the total clique size (V = 171, p = 0.0001069) were significantly lower when the bi-directed edges were removed from the pattern before converting it to a DAG and running the classification task.

Summarizing the results, it seems once again, that promoting sparseness, here by removing bi-directed edges, has few or no negative side effects (assuming that sparseness in networks is natural), and that we might be better off applying the heuristic liberally. However, removing bi-directed edges does not guarantee that we will obtain a tractable Bayesian network. If we revisit Figure 7.1, we see that besides bi-directed edges, nodes can still have large numbers of parents through "regular" directed arcs. Removing all the bi-directed edges

will not influence the set of already directed edges, and if a node has too many, the whole network will still be intractable. To solve this problem we will need to modify the pattern or DAG more rigorously, by limiting the number of parents a node is allowed to have. This is the subject of the next section.

7.4 LIMITING THE NUMBER OF PARENTS OF A NODE

To be able to provide a stronger guarantee for Bayesian network tractability, we need to constrain the CPTs of the BN. Given that the size of CPTs is exponential in the number of parents, the obvious fix for this problem is to limit the number of parents of a network. Faced with the problem that we have a parent set $\mathbf{Pa}(X)$ for a variable X, which exceeds the limit, we would like to select a subset $\mathbf{S}$, which contains the "optimal" parents and satisfies the size constraint. This additional step should promote network tractability while not negatively impacting the performance of the task, be it accurately representing the structure of a model or classifying samples.

I have tested four different measures that limit parent sets, among them an approach based on a Bayesian score similar to the parent limitation approach discussed by Silander and Myllymaki [2012]. In the remainder of this section, I will describe each of the approaches and will discuss the results of the two evaluation tasks.

All the approaches discussed in this chapter determine in one way or another the strength of the parent set. When we remove edges from the pattern, or DAG, we are in fact disregarding, in the case of the PC algorithm, statistical evidence for the existence of this edge (if we ignore the possibility of statistical mistakes). The compromise then should be that we remove the weakest edges. This way we prune the parent set, and keep the most relevant parent nodes.

7.4.1 P-Values

The first approach, applies the PC algorithm's independence testing procedure to estimate the strength of an edge. For each edge, connecting a node in the parent set to the child node, we perform a marginal independence test. We record the p-values of the tests and apply these as a measure of strength when pruning parents from the set. Smaller p-values are interpreted to indicate a stronger connection, i.e., the null hypothesis of independence would have been rejected more strongly. After testing all the edges of the parents, we rank them and prune the parent set to the top n parents. Here these are the parents with the smallest p-values. Unlike the PC algorithm, we do not apply the keep or the remove rule, since we are interested in the p-value. It does mean that when little data is available we will have less reliable information to work with, but we have to realize that the network structure itself was created with this same limited information as well, and the current situation leaves us most likely with an intractable Bayesian network unless we fix it by pruning edges.

7.4.2 Mutual Information

The second approach, is based on applying Mutual Information (MI) between nodes as a measure for strength. The MI approach is closely related to the previous p-value approach through the way they are calculated. For the p-value measure we calculate the G^2 statistic and determine the appropriate degrees of freedom for the χ^2 distribution, before plugging these values into the χ^2 distribution to obtain the p-value. As was mentioned before in Section 5.2.3, the mutual information measure is proportional to the G^2 statistic. I have reused the G^2 code to calculate the mutual information between a parent and a node of interest. Cheng et al. [1997], has applied mutual information as an alternative to independence tests in their TPDA algorithm. A larger MI value indicates a stronger connection between the nodes. The parent pruning procedure is, again, as follows: 1) We measure the MI between the node and all of its parents, 2) we rank the scores (this time larger is better), and 3) we prune the parent set to the top n scoring parents. This will guarantee a more tractable network through the reduced size of the child's CPT.

A difference between the p-value and the MI approaches is that the χ^2 calculation is more complicated, requiring not just the G^2 statistic but a degree of freedom calculation as well. The MI calculation is simpler, requiring the G^2 statistic, and the size of the dataset. When we have little data, this could work in our advantage, as we are not taking an unreliable measure and transforming it into one that might be even less reliable.

7.4.3 Bayesian Score

The final two approaches are both Bayesian of nature. Both calculate a Bayesian score, the BDeu score [Buntine, 1991], of the node given its parents, but differ in how they construct the optimal parent set. The first of the two, denoted in the empirical evaluation as *Bayes1*, follows closely the p-value and mutual information approaches by considering the parents of the node independently when scoring them. Bayesian scores are calculated for each parent, parents are ranked, and the top n parents remain after pruning the rest. The second approach, *Bayes2*, grows the optimal parent set similar to the Bayesian Search algorithm [Cooper and Herskovits, 1992] and the thickening phase of the Greedy-Thick-Thinning algorithm [Heckerman, 1995]. Starting with an empty parent set, we score all parents independently and add the best scoring parent to the final parent set. We repeat this procedure until we reach the preset parent limit size, but now the scores are based on the candidates for the final set plus the already confirmed parent nodes. Thus we run n rounds of Bayesian scoring to grow the best parent set.

7.4.4 Results

To test the performance of the different parent limitation approaches, I performed experiments with a gold standard recovery task and a classification accuracy task. The methodologies used for the two tasks are identical to the ones described in previous sections, with the single addition that I ran the experiment with three different parent limitation settings, $n = \{2, 4, 8\}$. Additionally to the four approaches, I added a random parent selection approach as a baseline. This approach randomly selects n parents and prunes the rest from the sets.

7.4.4.1 Experiment 7.3: Gold Standard Recovery Figure 7.5 shows the results of Experiment 7.3. The facets show for every dataset (the rows) and every parent limitation setting (the columns) the performance of the PC algorithm with either the keep or the remove rule and the five parent limitation approaches. I ran the parent limitation approaches on the keep rule result. The first column only contains the results of running the PC algorithm with the keep or remove rule without parent limitations. Especially for the larger networks, the Hamming distances for the keep rule in small dataset are so large, they effectively squelch all the details of the Hamming distance fluctuations of the other approaches. For clarity, the results from the remove rule are replicated in all columns.

If we inspect the results from the different approaches visually, we notice that the Bayes1 approach seems to produce patterns that consistently are more distant from the gold standard than the other approaches, especially when little data is available. The other approaches seem to be performing similarly to each other.

Unsurprisingly, in general, if the parent sets are constrained to a smaller number of parents, the Hamming distance tends to be lower, and if we increase the parent limit, the results from the five approaches will eventually converge on the original results from the pattern created using just the keep rule. Compared to the pattern created by the remove rule, the results are mixed. Depending on the specific dataset either the remove rule or the five parent limiting approaches would produce patterns with lower Hamming distances.

I performed a set of pairwise Wilcoxon signed rank tests, to find out if there were significant differences among the different parent limitation approaches and the keep,remove rule pair. The results are shown in Tables 7.1 and 7.2, where the former reports on two-tailed tests and the latter on one tailed tests.

Generally, there seems to be strong evidence that the different methods tend to produce statistically significant results. Table 7.1, shows the p-values of the two-tailed test. With the exception of three tests, all are significant (level of significance used : 0.05). It seems that the triplet of Pval, Bayes2, and Random cannot be distinguished from each other. To have an indication of which method would tend to produce patterns with lower Hamming distances than others, I performed one-tailed tests, where the alternative hypothesis would indicate that method x would create patterns with lower Hamming distances to the gold

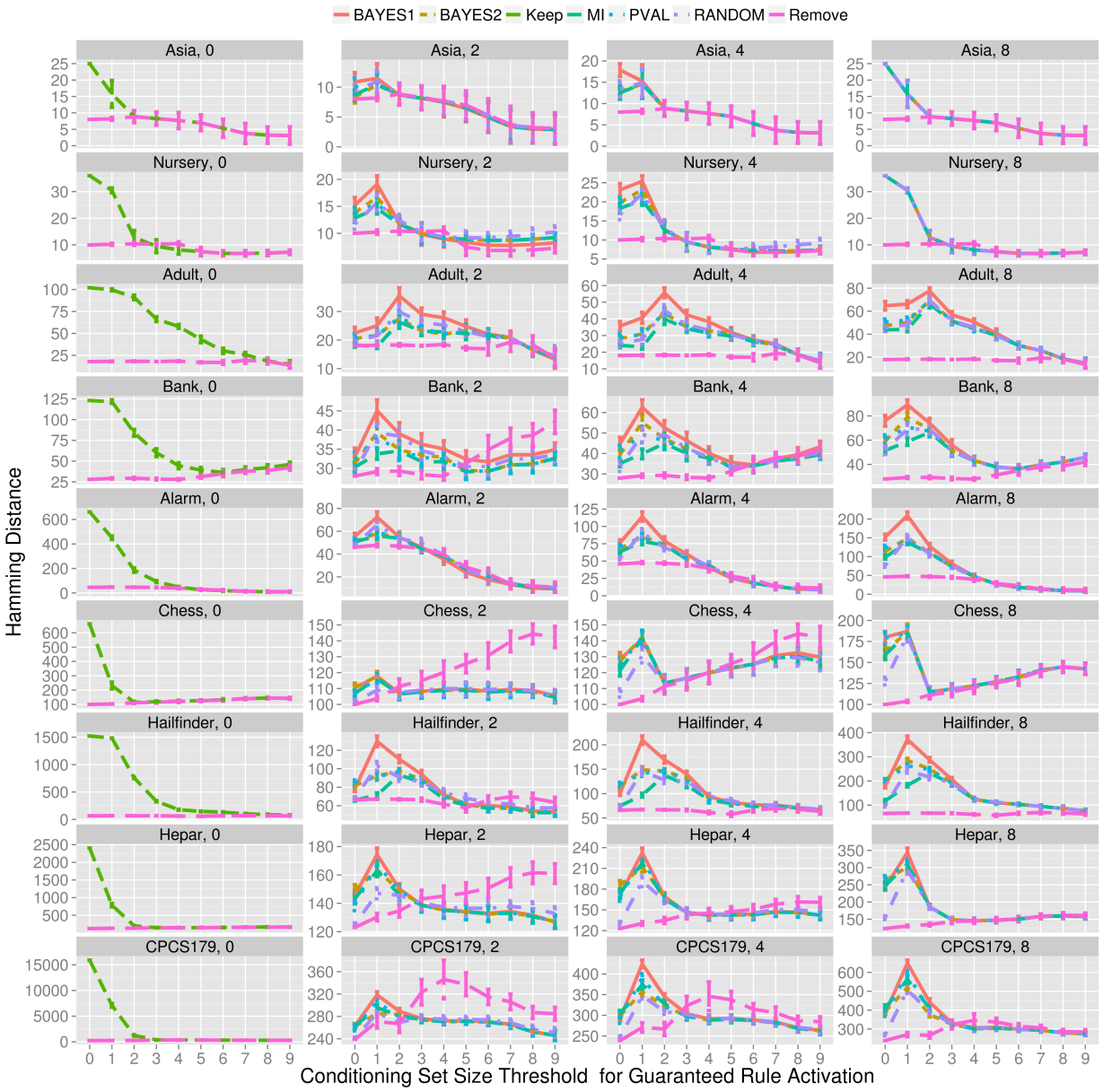


Figure 7.5: Gold Standard Recovery Results for Experiment 7.3

standard than method y. Table 7.2 shows the results.

From the results of the tests in the table we can conclude that applying just the remove rule instead of the keep rule followed by a parent limitation approach still tends to result in patterns which have lower Hamming distances with regard to the gold standard. Comparing the five parent limitation approaches, the approach based on the mutual information

Table 7.1: Gold Standard Recovery Two-Tailed Wilcoxon Signed Rank Test Results for Experiment 7.3 (Holm Corrected)

	Remove	Pval	MI	Bayes1	Bayes2	Random
Keep	< **2.2e-16**	< **2.2e-16**	< **2.2e-16**	< **2.2e-16**	< **2.2e-16**	< **2.2e-16**
Remove		**3.58e-05**	**1.603e-04**	**4.372e-06**	**3.582e-05**	**1.856e-05**
Pval			**6.089e-10**	< **2.2e-16**	0.2201	0.1007
MI				< **2.2e-16**	**2.265e-08**	**4.192e-06**
Bayes1					**5.796e-15**	**9.826e-06**
Bayes2						0.1315

Table 7.2: Gold Standard Recovery One-Tailed Wilcoxon Signed Rank Test Results for Experiment 7.3 (Holm Corrected)

$\downarrow x < y \rightarrow$	Keep	Remove	Pval	MI	Bayes1	Bayes2	Random
Keep	-	1	1	1	1	1	1
Remove	< **2.2e-16**	-	**4.477e-05**	**0.0002**	**4.129e-06**	**4.598e-05**	**2.121e-05**
Pval	< **2.2e-16**	1	-	1	< **2.2e-16**	1	0.2014
MI	< **2.2e-16**	1	**3.552e-10**	-	< **2.2e-16**	**1.96e-08**	**3.773e-06**
Bayes1	< **2.2e-16**	1	1	1	-	1	1
Bayes2	< **2.2e-16**	1	1	1	**3.344e-15**	-	0.3616
Random	< **2.2e-16**	1	1	1	**7.984e-06**	1	-

measures seems to be outperforming the others. The Bayes1 approach seems to perform the worst. For the other three, Pval, Bayes2, and the Random approaches, the evidence suggests that none of these three seems to be really outperforming the other, and it is important to note this includes the randomized parent selection approach only used as a baseline. It seems then that the MI approach is the preferred approach for the gold standard recovery task.

7.4.4.2 Experiment 7.4: Classification Accuracy Experiment 7.4 consisted of a classification task. The setup is identical to the classification task performed in previous sections (Experiments 5.2, 6.2, and 7.2), having only the parent restrictions added as an extra evaluation factor. Figure 7.6 shows the results of the experiment. It is difficult to visually make a distinction among the performances of the approaches. This was to be expected, since in the previous chapter we found no significant difference between the classification accuracies of networks created with the PC algorithm using the keep or remove rule. I

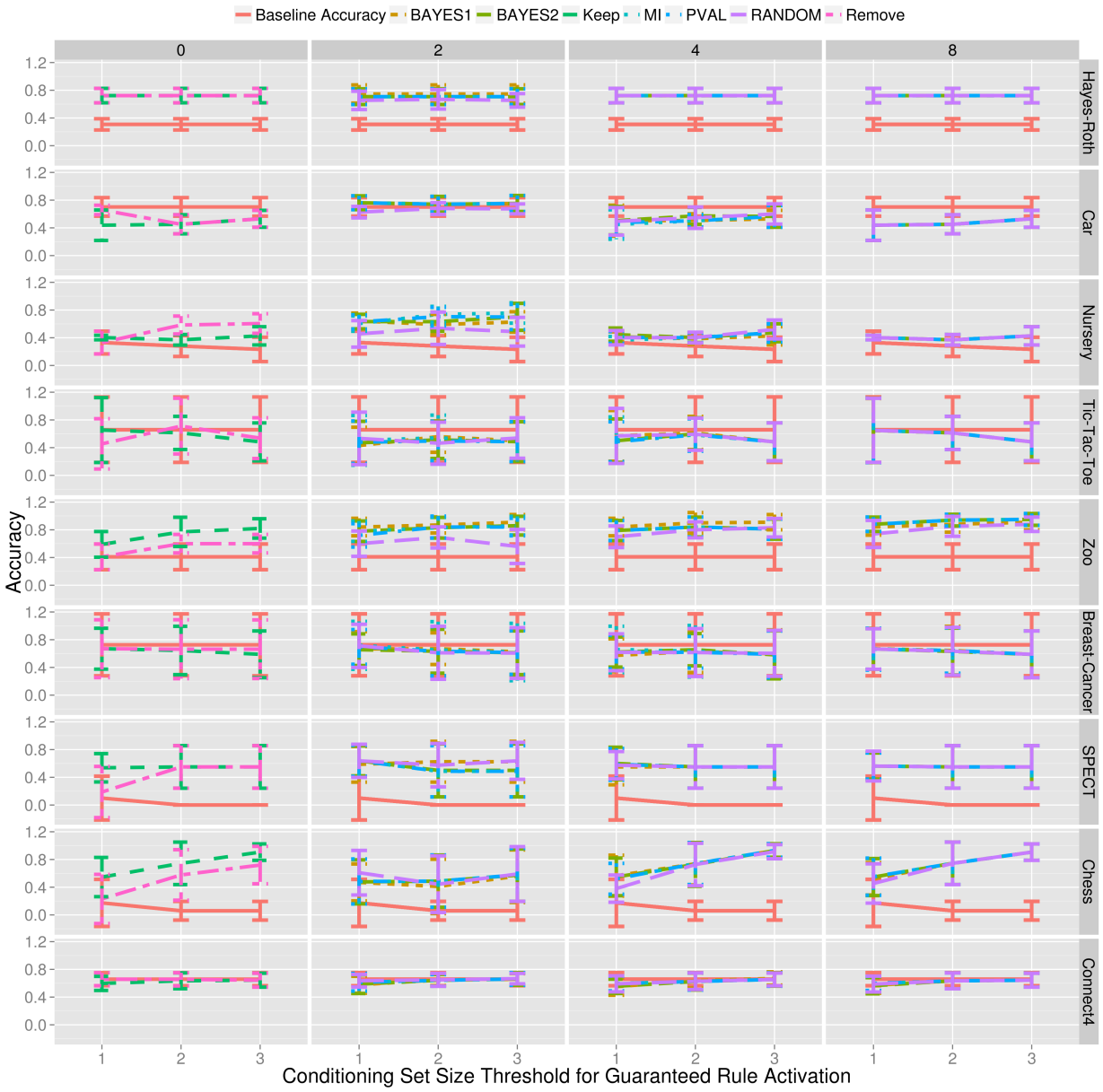


Figure 7.6: Average Classification Accuracy Results for Experiment 7.4

performed a closer examination using pairwise Wilcoxon tests. Table 7.3 shows the results of the two-tailed tests.

I found, after correcting the p-values using the Holm correction [Holm, 1979], that the classification accuracies, acquired by running the algorithm on three trimmed versions of the datasets and for three different parent limitation settings, that all approaches and rules

Table 7.3: Classification Accuracy Two-Tailed Wilcoxon Signed Rank Test Results for Experiment 7.4 (Holm Corrected)

	Remove	Pval	MI	Bayes1	Bayes2	Random	Baseline
Keep	0.7659	0.6275	0.5532	0.0866	0.7659	1	**0.0011995**
Remove		0.4411	0.3090	0.2793	0.4411	0.7659	**0.0046876**
Pval			0.7659	1	1	0.4411	**0.0002064**
MI				1	1	0.2830	**0.0002064**
Bayes1					1	0.2830	**0.0002064**
Bayes2						0.2914	**0.0002064**
Random							**0.0005651**

perform statistically different from the baseline. However, no significant differences were found between any of the approaches and rules. Table 7.4 shows that all approaches and rules perform better than the random baseline. After correcting for multiple tests, there is no statistical evidence for any approach or rule performing better than any other.

Table 7.4: Classification Accuracy One-Tailed Wilcoxon Signed Rank Test Results for Experiment 7.4 (Holm Corrected)

$\downarrow x > y \rightarrow$	Keep	Remove	Pval	MI	Bayes1	Bayes2	Random	Baseline
Keep	-	0.6960	1	1	1	1	1	**0.0005997**
Remove	1	-	1	1	1	1	1	**0.0023438**
Pval	0.6275	0.3729	-	1	1	1	0.2647	**0.0001032**
MI	0.5302	0.2511	0.8042	-	1	1	0.1564	**0.0001032**
Bayes1	0.0577	0.1885	1	1	-	1	0.1564	**0.0001032**
Bayes2	0.8047	0.4047	1	1	1	-	0.1628	**0.0001032**
Random	1	0.9166	1	1	1	1	-	**0.0002825**
Baseline	1	1	1	1	1	1	1	-

Although we could not find an improved classification performance after applying the approaches, the question is if these new network structures are more tractable than the original one. Figure 7.7 plots the classification accuracies and the total clique sizes. The values are normalized by the appropriate classification accuracy and total clique sizes of obtained from the keep heuristic. The upper-left quadrant, colored green, shows where the approaches are outperforming the keep heuristic, having better accuracy ($> 100\%$) and

smaller total clique size ($< 100\%$). With some exceptions, the methods either outperform or perform with little loss of accuracy while reducing the clique size significantly. This is backed

Figure 7.7: Total Clique Size vs Classification Accuracy

up statistically with the Wilcoxon test results for the total clique sizes. Table 7.5 shows the results for the two-tailed and one-tailed tests. All approaches differ from the results of the keep rule and have statistically significant smaller total clique sizes.

The results have shown that structure modifications can be useful to make the network more tractable, and sometimes improve its classification accuracy, but is there any way we

Table 7.5: Total Clique Size Wilcoxon Signed Rank Test Results for Experiment 7.4 (Holm corrected)

Test	Remove	Pval	MI	Bayes1	Bayes2	Random
Two-Tailed	**2.606e-10**	**2.464e-9**	**2.464e-9**	**5.66e-09**	**2.158e-9**	**1.850e-9**
One-Tailed: Keep $> x$	**1.304e-10**	**1.232e-9**	**1.232e-9**	**2.83e-09**	**1.0789e-10**	**9.249e-10**
One-Tailed: Keep $< x$	1	1	1	1	1	1

can do better? The next section investigates approaches that are meant to improve the classification performance of a Bayesian network.

7.5 STRUCTURE ADJUSTMENTS TO IMPROVE CLASSIFICATION ACCURACY

The last section showed that pruning the parent sets of nodes in a network can, in addition to making a Bayesian network more tractable, sometimes has a positive impact on the classification accuracy of the network. In this section, I explore three different heuristic approaches that aim to improve the classification accuracy of a Bayesian network. Similarly to the parent limiting approaches, and the bi-directed edges removal heuristic from the previous sections, the classification heuristics modify an existing structure. The three approaches do not use data to make the modifications, but make use of some general knowledge commonly used in algorithms for learning BN classifiers.

7.5.1 Markov Neighborhoods

The first approach applies a generalized version of the Markov blanket to prune edges from the network structure. The Markov blanket of a node consists of its parents, children and parents of its children. If all variables are observed, the node is conditionally independent of all other nodes. This is relevant when applying a regular Bayesian network to the classification problem. When classifying samples without missing data, the nodes in the Markov blanket

completely shield the class variable from the influences of all the other feature variables, making them irrelevant. If complete data is guaranteed, then we can prune the Bayesian network to just its Markov blanket without any negative consequences, but without this guarantee, the blanket may end up having a "hole" and useful information could flow into the class variable from other variables outside of the blanket.

One advantage of pruning the network this way is that we perform feature selection, we can completely remove nodes, and we will require fewer parameters for the BN.

For this approach I am applying a more general version of the Markov blanket, a *Markov neighborhood*, parameterized by an order n, to expand the original in size to include more variables that potentially can plug the holes in the original Markov blanket when data goes missing. My definition of a Markov neighborhood of order n for class variable C, $\mathbf{MN}(C, n)$, can be described recursively:

Definition 7.5.1 (Markov Neighborhood)**.**

$$\mathbf{MN}(C, 1) = \mathbf{MB}(C) , \tag{7.1}$$

$$\mathbf{MN}(C, n) = \bigcup_{i=\in \mathbf{MN}(C, n-1)} \mathbf{MN}(i, 1) , \tag{7.2}$$

where $\mathbf{MB}(X)$ is the Markov blanket of variable X. One can imagine that we "grow" the Markov neighborhood around the class variable C, starting with its Markov blanket and adding the Markov blankets of the blanket variables of C to the neighborhood. Eventually, if n grows large enough the Markov neighborhood will encompass the whole network. This means the Markov neighborhood approach will most likely be more useful for pruning larger networks than smaller ones. For smaller networks it will be more likely for the Markov neighborhood to contain the complete network.

7.5.2 Elevate Class Nodes

The Markov neighborhood approach only prunes the network. No changes are made to the edges that fall within the neighborhood. The main BN classifier algorithms such as naive Bayes, TAN, and ABN, make certain assumptions that simplify the structure and reduce the number of parameters. The naive Bayes classifier assumes that all feature variables

are conditionally independent of each other given the class variable. This is encoded in its distinctive topology.

The *elevate class nodes* heuristic, if we imagine the process visually, pulls the class variable, with its parents and children, up above all the variables and orients all its arcs away from the class node. This makes it parentless, similar to the popular BN classifiers, without influencing the rest of the network topology.

7.5.3 Augmented Bayesian Network

Finally, with the third approach, we modify the network structure more severely. We transform the original network into an Augmented Bayesian Network (ABN) classifier [Friedman et al., 1997]. We add arcs from the class node to all other feature variables, replacing any existing edges between the class node and feature variables.

7.5.4 Experiment 7.5: Performance of Classifier Enhancers

The methodology of Experiment 7.5, which evaluates the classifier enhancement functions, is similar to the previous classification task experiments (here referred to Experiment 7.5a), but it has an additional component, where I evaluate the classifier performance with missing values in the test data (referred to as 7.5b). I vary the percentage of missing variables in a sample and the percentage of samples with missing variables. Figure 7.8 shows the classification accuracies of the four approaches, the plain result of a network created with the PC algorithm using the remove rule, the result of a naive Bayes classifier used as a baseline, and the "very naive" baseline accuracy model that classifies solely based on the class proportions of the training data. A visual inspection of the results seems to indicate that the ABN approach and the NB classifier are the better performers in Experiment 7.5a. Similarly to the previous section, I tested the pairwise performance of the approaches using Wilcoxon tests, but in this case the results of these tests are not very reliable. Each test only had nine data points available to it, one per dataset. Tables 7.6 and 7.7 show no significant differences.

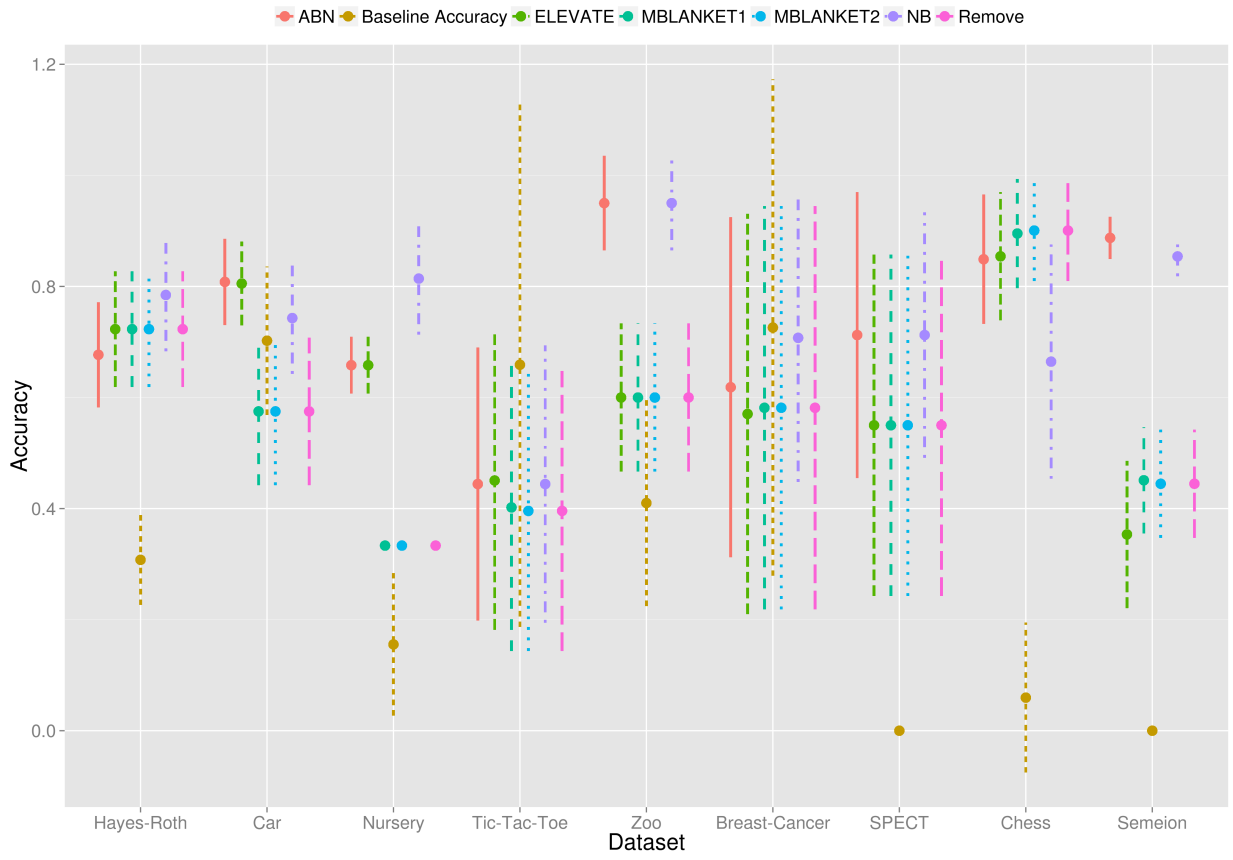


Figure 7.8: Average Classification Accuracy with Complete Data for Experiment 7.5

Table 7.6: Classification Accuracy Two-Tailed Wilcoxon Signed Rank Test Results using Complete Data for Experiment 7.5 (Holm Corrected)

	ABN	Elevate	MBlanket1	MBlanket2	NB	Baseline
Remove	0.9844	1	1	x	0.9844	1
ABN		1	0.9844	0.9844	1	0.7422
Elevate			1	1	1	0.9844
MBlanket1				1	0.9844	1
MBlanket2					0.9844	1
NB						0.5469

The second part of the evaluation, involving sixteen configurations of missing data, do provide us with sufficient data points to draw stronger statistical conclusions from the results. Figure 7.9 shows the results from this part of the evaluation. The main reason for

Table 7.7: Classification Accuracy One-Tailed Wilcoxon Signed Rank Test Results using Complete Data for Experiment 7.5 (Holm Corrected)

$\downarrow x > y \rightarrow$	Remove	ABN	Elevate	MBlanket1	MBlanket2	NB	Baseline
Remove	-	1	1	1	1	1	0.7813
ABN	0.5742	-	1	0.6309	0.5195	1	0.3906
Elevate	1	1	-	1	1	1	0.5195
MBlanket1	1	1	1	-	1	1	0.7813
MBlanket2	1	1	1	1	-	1	0.7813
NB	0.5742	1	1	0.5742	0.5742	-	0.2871
Baseline	1	1	1	1	1	1	-

evaluating the approaches with missing data is to examine the effect of the two versions of the Markov neighborhood approach. The evaluation includes a first and a second order version, named MBlanket1, and MBlanket2 respectively. The figure shows the performance of the approaches for the different datasets (columns), and how this (average classification accuracy) performance is affected by missing data. The x-axis of the facets shows the effect of more missing values per sample, and the rows of the figure show the effect of having more samples with missing values. Both quantify the number of missing values by a percentage (% of variables with missing values, vs % of missing samples).

In general, performance drops when more missing values are present in the sample. There are two cases where the classification accuracy actually improves with more missing data (Tic-Tac-Toe, Breast-Cancer), but here the different approaches are actually performing worse than the baseline accuracy model (shown in the figure as a thick straight red line), and, once all feature variables of a samples only consist of missing values, the models are reduced (or in these cases enhanced) to the level of the baseline classifier.

For most of the datasets, the performance of the different approach is very similar. In a few cases the ABN and NB approaches tend to stand out significantly from the rest. Since Experiment 7.5b has more data points than Experiment 7.5a, the results of the Wilcoxon tests to compare the performance of pairs of approaches were more reliable. Tables 7.8 and 7.9 show the results of the two-tailed and one-tailed tests respectively. The two tailed tests indicate that there is at least a statistically significant difference between most of the approaches. No significant difference was found between the performance of the original BN,

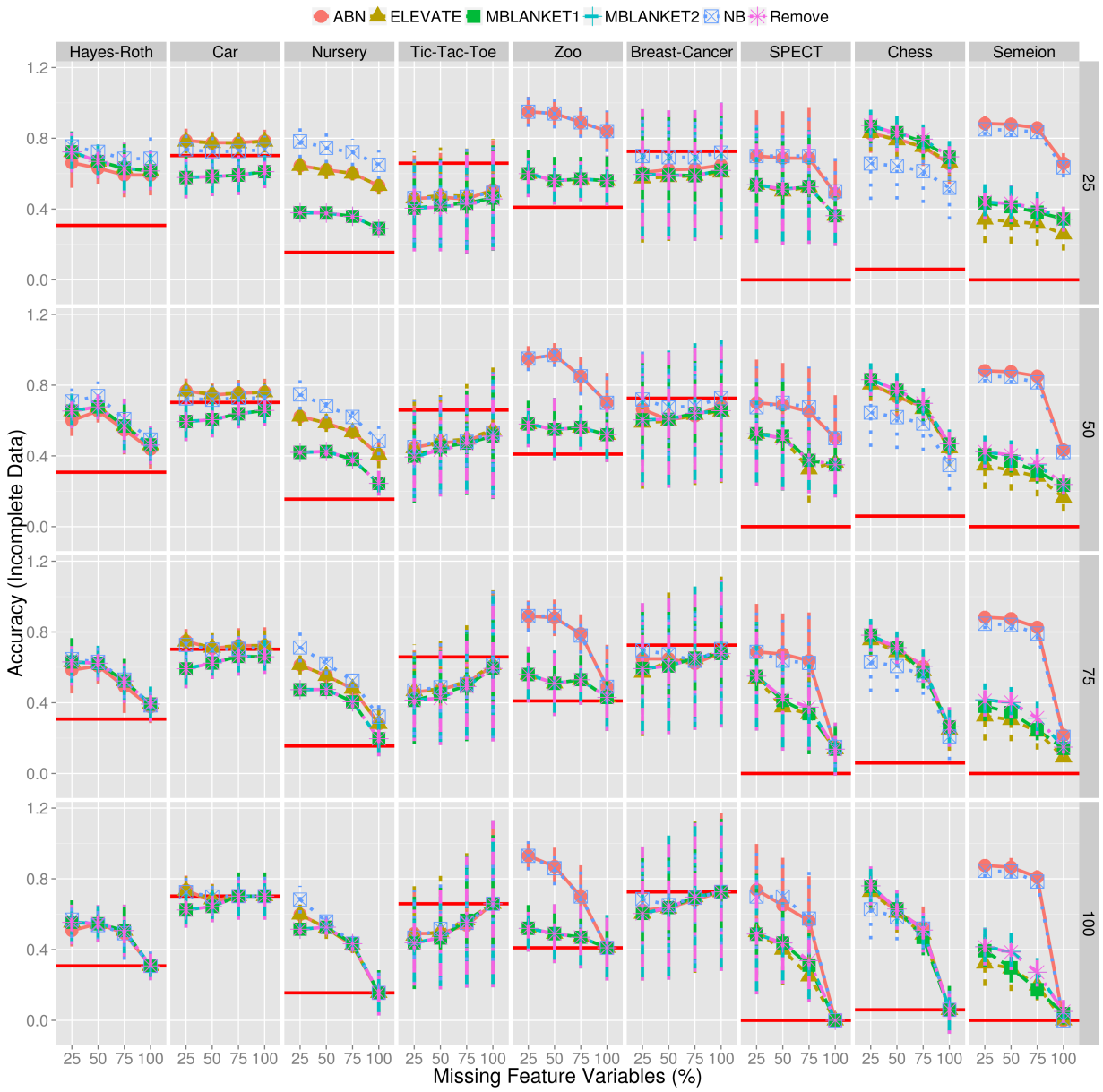


Figure 7.9: Average Classification Accuracy with Missing Data for Experiment 7.5

found using the PC algorithm with the remove rule, and the BN create by applying the Elevate or the MBlanket2 approach after the DAG conversion. Given that some of the test networks are smaller, it is possible that the second order Markov neighborhood encompasses the complete network, explaining the lack of difference between the performance of the two. Regarding the results of the Elevate approach, we can speculate that the structure changes

Table 7.8: Classification Accuracy Two-Tailed Wilcoxon Signed Rank Test Results using Incomplete Data for Experiment 7.5 (Holm Corrected)

	ABN	Elevate	MBlanket1	MBlanket2	NB
Remove	**1.7258e-13**	0.3768	**0.0173**	0.5989	**1.004e-13**
ABN		**1.213e-06**	**1.004e-13**	**1.726e-13**	0.5989
Elevate			0.0894	0.377	**4.336e-09**
MBlanket1				**0.0134**	**6.490e-14**
MBlanket2					**1.004e-13**

it performed on the original network were perhaps not so significant that it resulted in a markedly different performance. Additionally, the ABN and NB approaches do not seem to differ significantly from each other, but both do differ from the rest, as was observed visually from Figure 7.9.

Table 7.9: Classification Accuracy One-Tailed Wilcoxon Signed Rank Test Results using Incomplete Data for Experiment 7.5 (Holm Corrected)

$\downarrow x > y \rightarrow$	Remove	ABN	Elevate	MBlanket1	MBlanket2	NB
Remove	-	1	1	**0.0127**	1	1
ABN	**9.414e-14**	-	**9.854e-07**	**5.559e-14**	**1.098e-13**	1
Elevate	0.4304	1	-	0.0984	0.4710	1
MBlanket1	1	1	1	-	1	1
MBlanket2	1	1	1	**0.009559**	-	1
NB	**5.021e-14**	1	**2.650e-09**	**3.245e-14**	**5.021e-14**	-

The one-tailed test gives us a clear picture on which approaches, when looking at the big picture, perform the best. For both the ABN and NB approach there is statistically significant evidence that they outperform the others. There is quite strong statistical evidence that the MBlanket2 approach outperforms the MBlanket1 approach. This fits nicely with the theory; the network, pruned up to the second order Markov neighborhood of the class variable, surpassed the version that was pruned to only include the Markov blanket of the class variable at classifying samples, although neither method outperformed the original network. What was accomplished, is both resulted in networks with similar performance but with improved tractability.

7.6 DISCUSSION

In the empirical evaluations of the previous chapters, occasionally some of the networks structures became intractable after converting the patterns into complete Bayesian networks. Modifying the pattern or DAG structure before constructing the necessary conditional probability tables, is a potential remedy for this intractability problem. In this chapter, I have approached the intractability problem from two angles, and additionally, investigated if, with a similar methodology, making structural modifications can improve the classification accuracy of a model.

The first method is a heuristic that removes all bi-directed edges from the pattern before converting it to a Bayesian network. The results showed, for the datasets used in the evaluation, that removing bi-directed edges does not seem to have a significant (negative) effect on the performance of the gold standard recovery or sample classification task (although any change that promotes network sparsity tends to improve the gold standard recovery task result). However, it does not provide a definite solution for network intractability. If the pattern contains nodes that have a large parent set (through directed edges in the pattern), the network may still turn out to be intractable after conversion to the complete BN.

The second method, which sets a limit on the size of the parent set of a node, was extensively tested in the two-part empirical evaluation. This evaluation, consisting of the now familiar gold standard recovery task and classification task, pitted four different ways of determining the optimal parent set against each other, with a fifth approach that randomly assigned a set of parents to a node as a baseline. Limiting the parent sets improved the tractability of the network, decreased the Hamming distance in the gold standard recovery task, and, in general, did not affect the classification accuracy of networks in the classification task. From the four methods under investigation, the one based on the mutual information measure showed the best performance in the gold standard recovery task, and, without a clear winner in the classification task, seems to be the best quality measure choice when having to discard parents from the set.

Finally, following the promising results of the parent limitation approaches, I investigated the possibility of applying a similar method, making structural modifications to a network,

to improve the classification accuracy of networks. I investigated three different approaches, one of these was tested with two different parameter settings, and added a Naive Bayes classifier to the evaluation as an additional baseline to the default baseline used in this and previous evaluations. The results from the evaluation suggest that it is possible to improve the classification accuracy of a model by making structural changes, but the extend of the changes that were necessary to obtain this increase in accuracy, currently diminishes the usefulness of this approach.

The only approach that was statistically significant and consistently successful was the Augmented Bayesian network approach. This approach makes drastic changes to the original DAG. Arcs are added from the class node to all other nodes (reversing arcs that were pointing into the class node if they were present). Additionally, I found that the performance of the ABN adapted network structure did not differ in a statistically significant way from the naive Bayes classifier, meaning that we could safely dispose the remaining edges from the original Bayesian network structure without sacrificing performance.

We have to conclude that the proposed approaches are quite ineffective if a simple naive Bayes classifier can obtain the same results as the process involving 1) Learning a model using the PC algorithm, 2) transforming the pattern into a DAG, and 3) modifying the structure to improve the accuracy.

One positive result that I can report from this evaluation, is that the Markov neighborhood approach performed as expected. When classifying samples that contain missing data, the second order version of the MN structure modifier creates BN structures with higher classification accuracy than the structures created by the first order MN structure modifier.

To summarize, it is feasible, and frequently even beneficial, to modify the structure of a network to improve its tractability. My evaluation has shown that restricting the number of parents typically reduces the total clique size of the junction tree that is built from the Bayesian network when performing inference. In a considerable number of cases, the classification accuracy of the models either maintains or even increases the performance of the original network. Modifying network structures specifically to improve classification accuracy was successful, but, comes with a caveat. The only approaches that were showing an actual increase of performance over the original had made drastic changes to the original

structure to the point that the conditional (in)dependencies of the original model were no longer required to obtain the performance of the modified models. As potential future work I suggest replacing the approaches of Section 7.5, which are completely based on topology-driven modifications, with data driven approaches such as the ones discussed in Section 7.4.

8.0 CONCLUSIONS AND FUTURE RESEARCH

This chapter summarizes the work I have done for my contributions, if and how the results of my work support the hypotheses I proposed in the introduction and, finally, proposes future work.

8.1 CONCLUSIONS

In this dissertation, I have focused on several aspects of the constraint-based learning approach for learning Bayesian network structures. Initially, the focus of my work was one of large-scale learning, where I have investigated the potential of the MapReduce paradigm for learning BN structures. I have contributed two algorithms, and have compared these with two approaches from the literature and a single-computer baseline. As the second focus of my dissertation, I investigated what seems to be a misunderstanding in the literature on how to respond when there are insufficient data to test the independence of two variables. The literature suggests to either discard any edges which we cannot test, or conversely, to keep the edge, assuming the variables are dependent. I have investigated the impact of the two rules on the performance of two tasks. For the third and final focus of my dissertation, I addressed the problem of network tractability. An additional difficulty of applying the PC algorithm is that the end result typically is not a DAG, but a pattern. The pattern needs to be converted to a DAG before we can learn parameters and make use of the completed BN. Especially when less data is available, we frequently end up with very dense patterns and, eventually, DAG structures. I proposed and investigated five approaches to pruning a structure up to the point a tractable BN could be constructed. I followed up this work with

an experiment that tested the ability of a set of three approaches to improve the classification accuracy of a Bayesian network. In the introduction, I argued that my contributions all have a common theme. I postulated that design decisions for algorithms that learn or modify Bayesian network structures can positively influence the tractability and quality of Bayesian network inference and Bayesian network structure learning. Within this common theme, I put forward a hypothesis for each of the contributions. In the remainder of this section I will discuss the results of my work, the experiments I have performed, and if the data that I obtained from these experiments supports their respective hypothesis.

- *H1: It is computationally feasible to learn Bayesian network structures from datasets that have very large numbers of variables using the MapReduce framework.*

The results of the algorithm evaluation I performed support hypothesis H1. However, it is important to place the results into the proper context. The experiments utilized datasets with a relatively small number of records and mostly a small number of variables, with only a few datasets having hundreds and one dataset having thousands of variables.

The two algorithms from the literature [Chen et al., 2011, Fang et al., 2013], seem to have rather limited merit, looking at the results of the evaluation. Both require a large number of MapReduce jobs to complete the structure learning process, and this number depends polynomially on the number of variables, making it highly unlikely for the algorithms to scale to datasets with thousands of variables. However, the goal of both these algorithms was to allow for structure learning from datasets with many records, and it is reasonable to expect that in any case where these algorithms were able to successfully learn a network structure, they would still be able to do so if we increase the number of records of the datasets to sizes where cluster storage is required.

I proposed two MapReduce algorithms (PCA, PCB), both based on the PC algorithm, that avoided the issues of the literature algorithms with datasets with many variables, but which had their own problems. The PCA algorithm, similar to the two algorithms from the literature, parallelizes the calculations of sufficient statistics for the independence tests. In one Mapper phase, it calculates all statistics for all tests for a specified conditioning set size. This eventually always causes the MapReduce cluster to crash unless very specific conditions

are met. The PCB algorithm assumes that we can fit a copy of the complete dataset in the memory of every Mapper. Every Mapper then computes a chunk of all the independence tests, the Reduce phase is used to combine all the results. Assuming we can assign a sufficient amount of memory to each of the Mappers used by the algorithm, PCB performs the best of all four MapReduce algorithms, but for the majority of the datasets used for the evaluation, the single-computer PC algorithm dominated all of the MapReduce algorithms. Only when a dataset contained hundreds of variables or more, did the PCB algorithm complete the task faster than the single-processor baseline.

The results of the empirical evaluation show that, the networks created by the PCB algorithm are similar to the ones created by the PC algorithm (correctness), and datasets with a sufficient number of variables, are processed faster by the PCB algorithm than the PC algorithm (efficiency). On the classification task, the PCB algorithm did not perform as well as the PC algorithm, I suspect the partitioning strategy is to blame and, as a potential future direction of this work, a deeper investigation of the problem may be warranted. Assuming the data fits in memory, and that a sufficient amount of working memory is available to run the distributed independence tests in the Mappers, I believe that PCB can be scaled to learn models from datasets of at least tens of thousands of variables.

Within the boundaries of the evaluation, the single-computer baseline algorithm performed very well. If we are able to fit the data in memory, running the PC algorithm on a single computer might well be the best choice until the number variables in the datasets increase up to the point that the sheer number of independence tests that need to be performed becomes so large, that the overhead incurred by the PCB algorithm no longer is a significant portion of the total execution time. The experimental results indicate that this tipping point may be around 800 variables, but it will most likely depend on the specifics of the dataset such as the cardinalities of the variables. Similarly, if we scale the dataset in the number of records, at some point the algorithm by Chen et al. [2011] and Fang et al. [2013] will become preferable over a single-computer approach, unless we attempt to learn models using a dataset sampling scheme instead of the full dataset.

- *H2: Faced with an insufficient data sample to total contingency table cell number ratio, when testing the independence of two variables connected by an edge, removing this edge*

results in better Bayesian network structures.

The evaluation results support hypothesis H2. I compared the performance of both rules, keep [Spirtes et al., 1993], and remove [Tsamardinos et al., 2006], on a gold standard recovery (nine networks) and a classification task (nine datasets). The resulting networks differ the most from each other when little data are available when learning them. With sufficient data, the structural differences between the networks drop significantly and both end up performing similarly on the recovery task. For the classification task, over the nine datasets used for the evaluation, I found no statistically significant difference in the classification accuracy performance of the networks created using either rule. Additionally, I found no significant differences between the algorithm running time when learning the networks from data. I found a significant difference for the total clique sizes for the junction trees used for the inference and the time required to perform inference. In my experiments, I observed that it seems that networks created using the keep rule perform similarly to networks created using the remove rule. The exception occurred when we were recovering the gold standard structure from datasets of limited size. In this case the sparsity promoted by the remove rule resulted in networks that resembled the gold standard more closely than networks created using the keep rule. Thus, my experimental results indicate that, when we have sufficient data, we can be indifferent. However, the natural sparsity typically observed in Bayesian networks makes me believe that applying the remove rule will aid the PC algorithm in constructing structures that are sparser, and require less resources when performing inference.

- *H3: The tractability of inference of a Bayesian network can be improved by making structure modifications, without compromising its classification performance.*

The results of the evaluation support hypothesis H3. Removing bi-directed edges has merits, but cannot guarantee a tractable network. The results of the evaluation showed that by limiting the number of parents of the nodes we improved the tractability of inference for the resulting Bayesian networks. Since pruning edges promotes sparsity, we typically saw improvement in the gold standard recovery task. For the datasets used in the classification task, I found that statistically, there was no difference in the performance of the original and the pruned networks in terms of classification accuracy. I found that, for the datasets

that I considered in my experiments, pruned networks required less memory and performed inference, using SMILE's clustering algorithm implementation, quicker than the original network. Among the four parent limitation approaches, the approach based on the mutual information measure delivered the overall best performance when comparing the methods using the Wilcoxon signed rank test over the set of datasets used for the experiments.

- *H4: Well-chosen structure modifications can improve the classification accuracy of a Bayesian network.*

The results of the experiment support hypothesis H4. However, it must noted that the single strong positive result was obtained after the original network structure was modified drastically. Additionally, the performance of this standout approach (ABN) was similar to the performance of the naive Bayes classifier baseline, which seems to indicate that the edges from the original network had little to no effect on the classification accuracy of the modified network. My expectations for this work were that the other approaches would at least provide a marginal benefit, but the results of the evaluation did not support these expectations. There might be opportunities for improvement, and as potential future work data-driven methods should be investigated to replace the current, purely topological methods.

8.2 FUTURE RESEARCH

There are several opportunities for continuing the work that was done in this dissertation. Among them, a more extensive evaluation of the MapReduce algorithms with datasets that have more samples. This would be helpful to determine the domain where the algorithms of Chen et al. [2011] and Fang et al. [2013] are most useful compared to the other algorithms. Additionally this provides an opportunity to evaluate sampling schemes, which would be utilized to acquire representative samples of the datasets when they would be too large to be processed by PCB (not enough memory to fit in a Mapper), or by the "plain" PC algorithm (not enough memory to fit the data on one computer). This sampling procedure could be run multiple times to obtain multiple structures, which are then merged into the final result.

As was already mentioned in Chapter 5, a potential avenue of future research, involves investigating the inferior performance of the PCB algorithm on the classification task when partitioning the independence tests over many Mappers instances. A potential solution, which was proposed in the chapter, was to no longer make the partitions completely distinct, but to allow for partitions with a certain amount of overlap. This would result in the algorithm performing more work than necessary, but each of the partitions would now contains more information about the direct neighborhood of the edges we are testing. This extra information can impact the conditioning sets that the Mappers would use for the independence tests. Now, more Mappers might be aware that certain variables are independent of each other, and correctly eliminate these variables from each other's conditioning sets in future tests. This then reduces the size of the contingency table required for the independence tests, which might prevent the remove rule (as was described in Chapter 6) from triggering and unnecessarily removing edges, which possibly is causing the drop in performance that was observed in the evaluation.

Chapter 7 provides another opportunity for future work. Section 7.5 described approaches to improve the accuracy BN classifiers. With the exception of the ABN approach, all were unable to improve the classification accuracy of a network. Furthermore, for the datasets used in the experiment, the ABN approach made drastic changes to the original network structure and did not perform better than a naive Bayes classifier, making the arcs of the original network structure unnecessary. The methods proposed in section 7.5 only make changes based on topological principles. As future work, new methods should be proposed and evaluated that are based on a different principle such as the data-driven approaches that were used to limit the number of parents of nodes in section 7.4. As an example, a feature selection procedure could be adapted to, for every node, evaluate if adding extra children improves the classification accuracy. The initial starting point would be the class variable, after which the algorithm could work its way down through the DAG structure.

BIBLIOGRAPHY

J. Abellán, M. Gómez-Olmedo, and S. Moral. Some variations on the PC algorithm. In *Proceedings of the Third European Workshop on Probabilistic Graphical Models*, pages 1–8, 2006.

J. Abramson, B., W. Brown, A. Edwards, and R. Winkler Murphy. Hailfinder: A Bayesian system for forecasting severe weather. *International Journal of Forecasting*, 12(1):57–72, 1996.

Silvia Acid and Luis M. de Campos. Searching for Bayesian network structures in the space of restricted acyclic partially directed graphs. *Journal of Artificial Intelligence Research*, 18:445–490, 2003.

Davide Bacciu, Terence A Etchells, Paulo JG Lisboa, and Joe Whittaker. Efficient identification of independence networks using mutual information. *Computational Statistics*, pages 1–26, 2013.

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

Aniruddha Basak, Irina Brinster, Xianheng Ma, and Ole J Mengshoel. Accelerating bayesian network parameter learning using hadoop and mapreduce. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 101–108. ACM, 2012.

Ingo Beinlich, Jaap Suermondt, Martin Chavez, and Gregory Cooper. The ALARM Monitoring System: A Case Study with Two Probablistic Inference Techniques for Belief Networks. In *Second European Conference on Artificial Intelligence in Medicine*, pages 247–256, London, 1989.

Wray Buntine. Theory refinement on Bayesian networks. In *Proceedings of the Seventh conference on Uncertainty in Artificial Intelligence*, pages 52–60. Morgan Kaufmann Publishers Inc., 1991.

A. Cano, M. Gómez-Olmedo, and S. Moral. A score based ranking of the edges for the PC algorithm. In *Proceedings of the Fourth European Workshop on Probabilistic Graphical Models*, pages 41–48, 2008.

Causality Workbench Team. A phamacology dataset, 06 2008. URL http://www.causality.inf.ethz.ch/data/SIDO.html.

Wei Chen, Lang Zong, Weijing Huang, Gaoyan Ou, Yue Wang, and Dongqing Yang. An empirical study of massively parallel Bayesian networks learning for sentiment extraction from unstructured text. In *Web Technologies and Applications*, pages 424–435. Springer, 2011.

Jian Cheng and Marek J Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13(1):155–188, 2000.

Jie Cheng, David A Bell, and Weiru Liu. Learning belief networks from data: An information theory based approach. In *Proceedings of the sixth international conference on Information and knowledge management*, pages 325–331. ACM, 1997.

Andrii Cherniak, Huma Zaidi, and Vladimir Zadorozhny. Optimization strategies for a/b testing on hadoop. *Proc. VLDB Endow.*, 6(11):973–984, August 2013. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2536222.2536224.

David Maxwell Chickering. A transformational characterization of equivalent Bayesian network structures. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 87–98, San Francisco, CA, 1995. Morgan Kaufmann.

David Maxwell Chickering. Learning equivalence classes of Bayesian-network structures. *Journal of Machine Learning Research*, 2:445–498, 2002.

Francesco Colace, Massimo De Santo, Mario Vento, and Pasquale Foggia. Bayesian network structural learning from data: An algorithms comparison. In *ICEIS (2)*, pages 527–530, 2004.

Cristina Conati, Abigail S Gertner, Kurt VanLehn, and Marek J Druzdzel. On-line student modeling for coached problem solving using bayesian networks. *Courses And Lectures-International Centre For Mechanical Sciences*, pages 231–242, 1997.

Gregory F Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial intelligence*, 42(2):393–405, 1990.

Gregory F. Cooper and Edward Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.

Nicandro Cruz-Ramírez, Héctor-Gabriel Acosta-Mesa, Rocío-Erandi Barrientos-Martínez, and Luis-Alonso Nava-Fernández. How good are the Bayesian information criterion and the minimum description length principle for model selection? A Bayesian network analysis. In *MICAI*, pages 494–504, 2006.

Paul Dagum and Michael Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial intelligence*, 60(1):141–153, 1993.

Denver Dash and Gregory F Cooper. Exact model averaging with naive bayesian classifiers. In *ICML*, pages 91–98, 2002.

Denver Dash and Marek J. Druzdzel. A hybrid anytime algorithm for the construction of causal models from sparse data. In *Proceedings of the Fifteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pages 142–149, 1999. URL citeseer.ist.psu.edu/dash99hybrid.html.

Cassio P De Campos, Zhi Zeng, and Qiang Ji. Structure learning of bayesian networks using constraints. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 113–120. ACM, 2009.

Martijn de Jongh and Marek Druzdzel. A comparison of structural distance measures for causal Bayesian network models. In *Proceedings of the International Joint Conference on Intelligent Information Systems*. IEEE Computer Society, 2009.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI 2004*, pages 137–150, 2004. URL http://www.usenix.org/events/osdi04/tech/dean.html.

Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, 7:1–30, December 2006. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=1248547.1248548.

Dorit Dor and Michael Tarsi. A simple algorithm to construct a consistent extension of a partially oriented graph. *Technicial Report R-185, Cognitive Systems Laboratory, UCLA*, 1992.

Qiyu Fang, Kun Yue, Xiaodong Fu, Hong Wu, and Weiyi Liu. A MapReduce-based method for learning Bayesian network from massive data. In *Web Technologies and Applications*, pages 697–708. Springer, 2013.

Stephen E Fienberg. *The analysis of cross-classified categorical data*. Springer, 2007.

Stephen E Fienberg and Paul W Holland. Methods for eliminating zero counts in contingency tables. *Random Counts on Models and Structures*, pages 233–260, 1970.

Nir Friedman and Moises Goldszmidt. Building classifiers using bayesian networks. In *Proceedings of the national conference on artificial intelligence*, pages 1277–1284, 1996.

Nir Friedman and Daphne Koller. Being Bayesian about network structure. In *Machine Learning*, pages 201–210, 2000.

Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

Robert Fung and Brendan Del Favero. Backward simulation in Bayesian networks. In *Proceedings of the Tenth international conference on Uncertainty in Artificial Intelligence*, pages 227–234. Morgan Kaufmann Publishers Inc., 1994.

Robert M Fung and Kuo-Chu Chang. Weighing and integrating evidence for stochastic simulation in bayesian networks. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 209–220. North-Holland Publishing Co., 1990.

John Geweke. Bayesian inference in econometric models using Monte Carlo integration. *Econometrica: Journal of the Econometric Society*, pages 1317–1339, 1989.

William D Gropp, Ewing L Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. the MIT Press, 1999.

David Heckerman. A tutorial on learning with Bayesian networks. Technical report, Learning in Graphical Models, 1995.

David Heckerman, Dan Geiger, and David M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

Max Henrion. Propagation of uncertainty by probabilistic logic sampling in Bayes' networks. In *Uncertainty in Artificial Intelligence*, volume 2, pages 149–164, 1988.

Luis D Hernandez, Serafin Moral, and Antonio Salmeron. A Monte Carlo algorithm for probabilistic propagation in belief networks based on importance sampling and stratified simulation techniques. *International Journal of Approximate Reasoning*, 18(1):53–91, 1998.

Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *Int. J. Approx. Reasoning*, 15(3):225–263, 1996.

Jaime Ide and Fabio G. Cozman. Random generation of Bayesian networks. In *in Brazilian Symp. on Artificial Intelligence*, pages 366–375. Springer-Verlag, 2002.

Jin H Kim and Judea Pearl. A computational model for causal and diagnostic reasoning in inference systems. In *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 1*, pages 190–193. Morgan Kaufmann Publishers Inc., 1983.

Wai Lam. Bayesian network refinement via machine learning approach. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(3):240–251, 1998.

Wai Lam and Alberto Maria Segre. A distributed learning algorithm for Bayesian inference networks. *Knowledge and Data Engineering, IEEE Transactions on*, 14(1):93–105, 2002.

Steffen L Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics & Data Analysis*, 19(2):191–201, 1995.

Steffen. L. Lauritzen and David. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50:157–224, 1988.

Long Liu, Wei Hu, Chunrong Lai, Hong-shan Jiang, Wenguang Chen, Weimin Zheng, and Yimin Zhang. Parallel module network learning on distributed memory multiprocessors. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 129–134. IEEE, 2005.

Nam Ma, Yinglong Xia, and Viktor K Prasanna. Parallel exact inference on multicore using mapreduce. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 187–194. IEEE, 2012.

G. Melancon, I. Dutour, and M. Bousquet-Mlou. Random generation of DAGs for graph drawing. Technical report, Centre for Mathematics and Computer Sciences, Amsterdam INS-R0005, 2000.

Alexander Mendiburu, José Miguel-Alonso, and Jose Antonio Lozano. Implementation and performance evaluation of a parallelization of estimation of Bayesian network algorithms. *Parallel Processing Letters*, 16(01):133–148, 2006.

Thomas Mensink, Wojciech Zajdel, and Ben Krose. Distributed em learning for appearance based multi-camera tracking. In *Distributed Smart Cameras, 2007. ICDSC'07. First ACM/IEEE International Conference on*, pages 178–185. IEEE, 2007.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953.

Stefano Monti and Gregory F. Cooper. Learning Bayesian belief networks with neural network estimators. In *Neural Information Processing Systems 9*, pages 579–584. MIT Press, 1997.

Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *J. Artif. Int. Res.*, 8(1):67–91, March 1998. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=1622788.1622792.

Masaharu Munetomo, Naoya Murao, and Kiyoshi Akama. Empirical studies on parallel network construction of Bayesian optimization algorithms. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1524–1531. IEEE, 2005.

Kevin P Murphy, Yair Weiss, and Michael I Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on*

Uncertainty in Artificial Intelligence, pages 467–475. Morgan Kaufmann Publishers Inc., 1999.

Vasanth Krishna Namasivayam and Viktor K Prasanna. Scalable parallel implementation of exact inference in bayesian networks. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 8–pp. IEEE, 2006.

Julian R Neil, Chris S Wallace, and Kevin B Korb. Learning bayesian networks with restricted causal interactions. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 486–493. Morgan Kaufmann Publishers Inc., 1999.

Olga Nikolova and Srinivas Aluru. Parallel discovery of direct causal relations and Markov boundaries with applications to gene networks. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 512–521. IEEE, 2011.

Olga Nikolova, Jaroslaw Zola, and Srinivas Aluru. A parallel algorithm for exact Bayesian network inference. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 342–349. IEEE, 2009.

Jiří Očenášek and Josef Schwarz. The parallel Bayesian optimization algorithm. In *The State of the Art in Computational Intelligence*, pages 61–67. Springer, 2000.

Agnieszka Onisko. *Probabilistic Causal Models in Medicine: Application to Diagnosis of Liver Disorders.* PhD thesis, Institute of Biocybernetics and Biomedical Engineering, Polish Academy of Science, Warsaw, March 2003.

Luis Ortiz and Leslie Kaelbling. Adaptive importance sampling for estimation in structured domains. In *Proceedings of the Sixteenth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-00)*, pages 446–454, San Francisco, CA, 2000. Morgan Kaufmann.

Judea Pearl. Evidential reasoning using stochastic simulation of causal models. *Artificial Intelligence*, 32(2):245–257, 1987.

Judea Pearl. *Probabalistic Reasoning in Intelligent Systems*. Morgan Kaufman, San Mateo, 1988.

Judea Pearl and Thomas S. Verma. A theory of inferred causation. In J.A. Allen, R. Fikes, and E. Sandewall, editors, *KR–91, Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pages 441–452, Cambridge, MA, 1991. Morgan Kaufmann Publishers, Inc., San Mateo, CA.

David M Pennock. Logarithmic time parallel bayesian inference. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 431–438. Morgan Kaufmann Publishers Inc., 1998.

Eric Perrier, Seiya Imoto, and Satoru Miyano. Finding optimal Bayesian network given a super-structure. *Journal of Machine Learning Research*, 9:2251–2286, 2008.

Malcolm Pradhan, Gregory Provan, Blackford Middleton, and Max Henrion. Knowledge engineering for large belief networks. In *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*, pages 484–490. Morgan Kaufmann Publishers Inc., 1994.

Parot Ratnapinda and Marek J. Druzdzel. An empirical comparison of Bayesian network parameter learning algorithms for continuous data streams. In *Recent Advances in Artificial Intelligence: Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS–2013)*, pages 627–632, 2013.

Teemu Roos, Hannes Wettig, Peter Grünwald, Petri Myllymäki, and Henry Tirri. On discriminative bayesian network classifiers and logistic regression. *Machine Learning*, 59(3): 267–296, 2005.

R.Y. Rubinstein. *Simulation and the Monte Carlo method*. John Wiley & Sons, 1981.

Ferat Sahin and Archana Devasia. Distributed particle swarm optimization for structural bayesian network learning. *Swarm Intelligence: Focus on Ant and Particle Swarm Optimization*, 27:505–532, 2007.

Wesley C. Salmon. *Scientific Explanation and the Causal Structure of the World*. Princeton University Press, Princeton, NJ, 1984.

Ross Shachter and Mark Peot. Simulation approaches to general probabilistic inference on belief networks. In *Proceedings of the Fifth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-89)*, pages 311–318, Corvallis, Oregon, 1989. AUAI Press.

Ross D Shachter and Mark A Peot. Simulation approaches to general probabilistic inference on belief networks. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 221–234. North-Holland Publishing Co., 1990.

Michael Shwe and Gregory Cooper. An empirical analysis of likelihood-weighting simulation on a large, multiply connected medical belief network. *Computers and Biomedical Research*, 24(5):453–475, 1991.

Tomi Silander and Petri Myllymaki. A simple approach for finding the globally optimal bayesian network structure. *arXiv preprint arXiv:1206.6875*, 2012.

Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, Prediction, and Search*. MIT Press, Cambridge, Massachusetts, 1st edition, 1993.

Peter Spirtes, Clark Glymour, and Richard Scheines. *Causation, Prediction, and Search*. MIT Press, 2. ed edition, 2000. ISBN 0-262-19440-6. URL http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+312309759&sourceid=fbw_bibsonomy.

Robert Stojnic, Audrey Qiuyan Fu, and Boris Adryan. A graphical modelling approach to the dissection of highly correlated transcription factor binding site profiles. *PLoS computational biology*, 8(11):e1002725, 2012.

Yoshinori Tamada, Seiya Imoto, and Satoru Miyano. Parallel algorithm for learning optimal Bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459, 2011.

Ioannis Tsamardinos, Laura E. Brown, and Constantin F. Aliferis. The max-min hill-climbing Bayesian network structure learning algorithm. *Mach. Learn.*, 65(1):31–78, 2006.

Thomas Verma and Judea Pearl. Equivalence and synthesis of causal models. In *UAI '90: Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence*, pages 255–270, New York, NY, USA, 1991. Elsevier Science Inc.

Michael P Wellman. Fundamental concepts of qualitative probabilistic networks. *Artificial Intelligence*, 44(3):257–303, 1990.

Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1(6):80–83, 1945.

Yinglong Xia and Viktor K Prasanna. Junction tree decomposition for parallel exact inference. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

Yang Xiang and Tongsheng Chu. Parallel learning of belief networks in large and difficult domains. *Data Mining and Knowledge Discovery*, 3(3):315–339, 1999.

Kui Yu, Hao Wang, and Xindong Wu. A parallel algorithm for learning Bayesian networks. In *Advances in Knowledge Discovery and Data Mining*, pages 1055–1063. Springer, 2007.

Changhe Yuan and Marek J Druzdzel. Importance sampling algorithms for Bayesian networks: Principles and performance. *Mathematical and Computer Modelling*, 43(9):1189–1207, 2006.